313-CD-001-002

EOSDIS Core System Project

EOSDIS Core System (ECS) Internal Interface Control Document for the Flight Operations Segment

October 1995

Hughes Information Technology Corporation
Upper Marlboro, MD

EOSDIS Core System (ECS) Internal Interface Control Document for the Flight Operations Segment

October 1995

Prepared Under Contract NAS5-60000
CDRL Item 051

SUBMITTED BY

Cal E. Moore, Jr. /s/ 9/29/95

Cal Moore, FOS CCB Chairman Date
EOSDIS Core System Project

Hughes Information Technology Corporation
Upper Marlboro, Maryland

This page intentionally left blank.

Preface

Object models presented in this document have been exported directly from CASE tools and in some cases contain too much detail to be easily readable within hard copy page constraints. The reader is encouraged to view these drawings on line using the Portable Document Format (PDF) electronic copy available via the ECS Data Handling System (EDHS) at URL http:// edhs1.gsfc.nasa.gov.

This document defines the interface between the Flight Operations Segment (FOS) and the Science Communications and Development Organization (SCDO). In particular it defines the SCDO Services that FOS will use in the design and how they interface with the Flight Operations Segment.

This document is a contract deliverable with an approval code 2. As such, it does not require formal Government approval, however, the Government reserves the right to request changes within 45 days of the initial submittal. Once approved, contractor changes to this document are handled in accordance with Class I and Class II change control requirements described in the EOS Configuration Management Plan, and changes to this document shall be made by document change notice (DCN) or by complete revision.

Any questions should be addressed to:

Data Management Office
The ECS Project Office
Hughes Information Technology Corporation
1616 McCormick Drive
Upper Marlbor, MD 20774

This page intentionally left blank.

Abstract

This document defines the interface between the Flight Operations Segment (FOS) and the Science Communications and Development Organization (SCDO). In particular it defines the SCDO Services that FOS will use in the design and how they interface with the Flight Operations Segment.

Keywords: FOS, design, specification, analysis, IST, EOC, interface

This page intentionally left blank.

Change Information Page

List of Effective Pages	
Page Number	**Issue**
Title	Final
iii through xii	Final
1-1 and 1-2	Final
2-1 and 2-2	Final
3-1 through 3-96	Final
4-1 through 4-26	Final
5-1 through 5-20	Final
A-1 and A-2	Final
B-1 through B-8	Final
AB-1	Final

Document History			
Document Number	**Status/Issue**	**Publication Date**	**CCR Number**
313-CD-001-001	Preliminary	January 1995	95-0007
313-CD-001-002	Final	October 1995	95-0648

This page intentionally left blank.

Contents

4. MSS Services

5. SDPS Services

Figures

Tables

Appendix A. MSS Managed Hardware Objects

Appendix B. MSS Managed Application (Software) Objects

Abbreviations and Acronyms

1. Introduction

1.1 Identification

The contents of this document defines the ECS Internal Interface Control Document for the Flight Operations Segment (FOS). Thus, this document addresses the Data Item Description (DID) for CDRL item 051 313/DV2 under Contract NAS5-60000.

1.2 Scope

This document defines the interface between the Flight Operations Segment (FOS) and the Science Communications and Development Organization (SCDO). In particular it defines the SCDO Services that FOS will use in the design and how they interface with the Flight Operations Segment.

This document reflects the August 23, 1995 Technical Baseline maintained by the contractor configuration control board in accordance with ECS Technical Direction No. 11, dated December 6, 1994.

1.3 Purpose

The purpose of this document is to define the interface between FOS and SCDO.

1.4 Status and Schedule

This submittal of DID 313/DV2 incorporates the FOS/SCDO interface detailed design performed during the Critical Design Review (CDR) time frame. This document is under the ECS Project configuration control.

1.5 Document Organization

Abbreviations and acronyms contains an alphabetized list of the definitions for abbreviations and acronyms used within this design specification.

This page intentionally left blank.

2. Related Documentation

2.1 Parent Documents

The following documents are the parents from which this document's scope and content derive:

304-CD-001-002	Flight Operations Segment (FOS) Requirements Specification for the ECS Project, Volume 1: General Requirements
304-CD-002-002	EOSDIS Core System Project, Science Data Processing Segment (SDPS) Requirements Specification for the ECS Project
304-CD-003-002	Communications and System Management Segment (CSMS) Requirements Specification for the ECS Project
304-CD-004-002	Flight Operations Segment (FOS) Requirements Specification for the ECS Project, Volume 2: AM-1 Mission Specific
423-41-02	Goddard Space Flight Center, Functional and Performance Requirements Specification for the Earth Observing System Data and Information System (EOSDIS) Core System

2.2 Applicable Documents

The following documents are referenced herein and are directly applicable to this document. In the event of conflict between any of these documents and this document, this document shall take precedence.

305-CD-001-002/ 311-CD-001-002	Flight Operations Segment (FOS) Design Specification and FOS Database Design and Database Schema Specifications
305-CD-002-002	EOSDIS Core System Project, Science Data Processing Segment (SDPS) Design Specification for the ECS Project (under development)
305-CD-003-002	Communications and System Management (CSMS) Design Specification for the ECS Project

2.3 Information Documents

The following documents, although not directly applicable, amplify or clarify the information presented in this document, but are not binding.

311-CD-001-003	Flight Operations Segment (FOS) Database Design & Database Schema for the ECS Project

This page intentionally left blank.

3. Communications Subsystem (CSS) Services

3.1 CSS Overview

The Communications Subsystem (CSS) provides for the interconnection of users and service providers, transfer of information within the Earth Observing System Data Information System (EOSDIS) Core System (ECS) and between ECS and many EOSDIS components, and management of all ECS communications components. It supports and interacts with the Science Data Processing Segment (SDPS) and the Flight Operations Segment (FOS). This section provides an overview and critical design characterization of CSS services required for FOS, its users and operators, and the Management subsystem (MSS - refer to Section 4 of this document).

The CSS provides ECS applications, including all FOS, SDPS, and CSMS components, with services for distributed communications. The CSS is composed of a combination of Commercial Off The Shelf (COTS), customized public domain software and custom applications to provide a highly automated and integrated means for providing these distributed processing capabilities to the various ECS applications and ECS users. This section describes the use of CSS services to provide distributed communications capabilities to FOS applications, located at the EOS Operations Center (EOC) and ISTs, and other FOS users. FOS applications communicating with the ECS Management Subsystem (MSS) also use these CSS distributed services.

The CSS services and interfaces have been described in detail in the following documents:

- 305-CD-012-001 Release-A CSMS Communications Subsystem Design Specification for the ECS Project;
- 313-CD-004-001 Release A CSMS/SDPS Internal Interface Control Document for the ECS Project; and
- 542-TP-003-001 Release A Communications and Management Subsystems Developer's Guide for the ECS Project.

In the rest of this section only CSS services and interfaces that are directly required and used by FOS applications and their users are addressed. Please refer to the above mentioned documents for complete details on the CSS services and their interfaces.

3.1.1 CSS Context

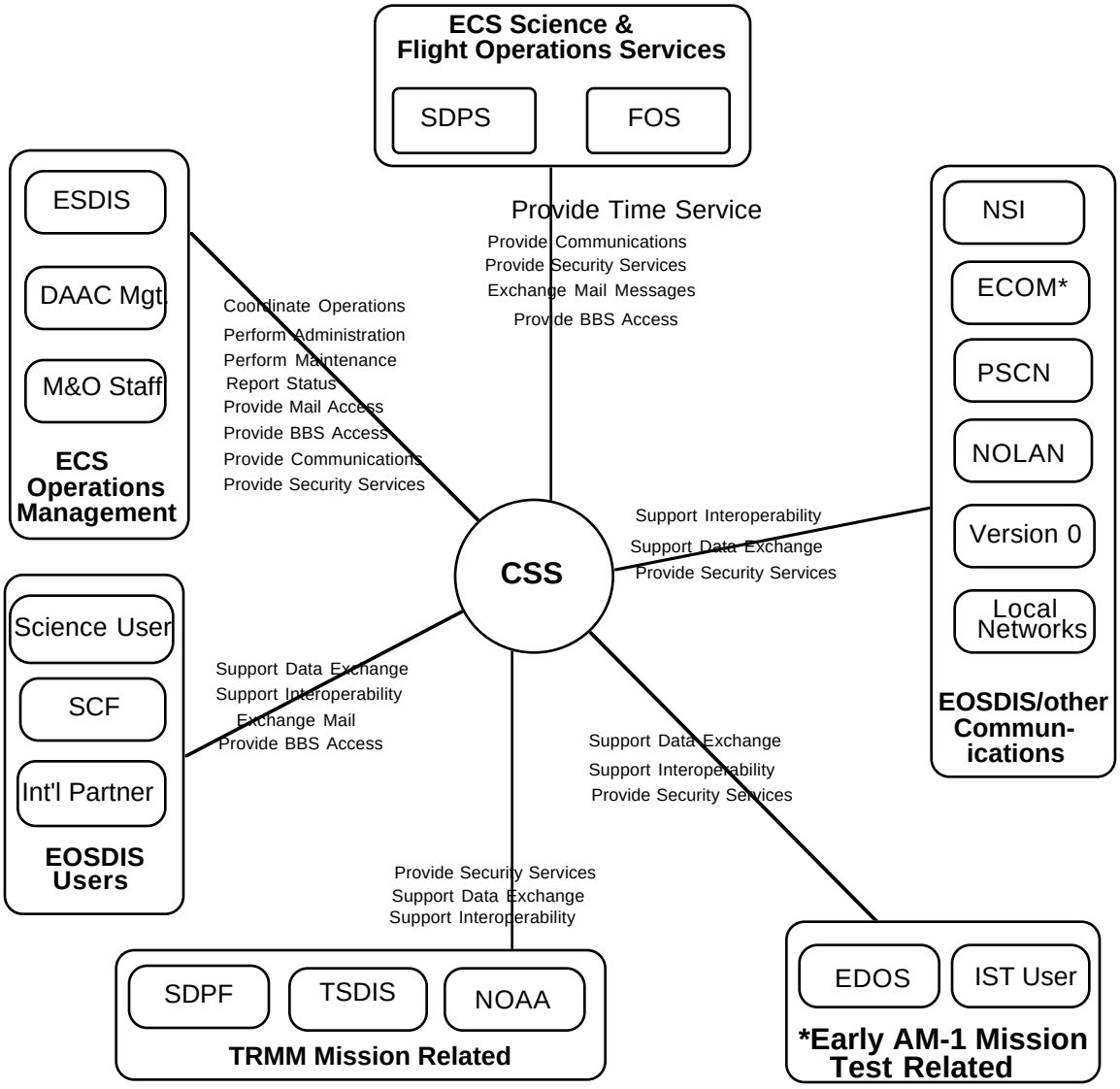

Figure 3.1.1-1. CSS Context

CSS is distributed across all ECS components. On client and server platforms, the CSS provides FOS applications with access to legacy services such as mail, bulletin board, file transfer and host access as well as object-oriented infrastructure services upon which to execute client-server operations. Client platforms outside the ECS installation (e.g., ISTs) are provided with a subset of CSS services which are integrated within the ECS Toolkit software. In addition to installation on SDPS and FOS platforms, CSS services are also installed on CSS and MSS servers and workstations distributed throughout the ECS.

Through the NASA Science Internet (NSI) and local institutional network connectivity, CSMS supports FOS service access by EOSDIS user communities, including IST operations. These interfaces facilitate exchange of various data items (e.g., science products, algorithms, ancillary data sets) as well as interactive access sessions with FOS and CSMS management services. Within ECS, CSMS communications services are provided to support FOS applications and infrastructure to facilitate peer-to-peer, client-server communication including required value-added services.

A brief summary of each of the CSCIs is described here. A more detailed version of th0se services that directly affect FOS software development are presented in more detail later in this section.

NOTE: Usage of the CSS services is based on requirements of the FOS applications for distributed communications. At a minimum a client-server or peer-to-peer application will use the CSS directory/naming service and security service at the initiation of a session. During the initiation of the session the application specifies the method of communication (i.e., IPC/HCL, message passing, RPCs, sockets) to be used and the level of security required.

Directory Naming Service

The Directory Naming Service provides a reliable mechanism by which distributed applications can associate information with names. Its primary purpose is to allow clients to locate servers. Its capabilities, however, are general-purpose, and it can be used in any application that needs to make names and their attributes available throughout a network.

CSS will provide implementation of both the DNS and the X.500 by supporting BIND and OSF Global Directory Service and OSF Cell Directory Service (CDS). It also provides application programmers the ability to store, retrieve, list information in the locally supported namespaces. The DNS and X.500 namespaces are used to connect the locally supported CDS namespaces. The functionality provided here will be implemented on top of XDS/XOM interfaces. As such, application programmers can use the above mentioned services (store, retrieve, list) in CDS as well as OSF GDS.

Security Service

The security service provides secure transfer of data on local and wide area networks. It provides mechanisms to verify the identity of users, and to determine whether users are permitted to invoke certain operations (authentication and authorization). Transmission of data is protected through the use of checksums and encryption of data. Authentication is provided by trusted third party (secret key) authentication. Authorization is based on Access Control Lists. The protocol used for authentication is Kerberos. All of these features are implemented within the ECS domain by employing OSF/DCE Security Services.

Multicast

Multicasting is a mechanism through which a single copy of data is transferred from a single point to several places. Multicasting allows a sending application to specify a multicast address and send one copy of the data to that address. This data is then distributed through the Multicast backbone to all the applications listening at that address. This reduces the network traffic and improves the performance.

Multicast is being used by FOS as a Release B service. Descriptions of this service are presented in Section 3.1.2.

Message Passing Service

The Message Passing Service allows for the exchange of information between applications running on different platforms. Clients send data to servers, which process the data and return the result back to the client. This interaction can be classified into three categories: synchronous, asynchronous, and deferred synchronous.

CSS will provide two implementations of Message Passing. The first model will provide for asynchronous and synchronous message passing - byte streams only - with store and forward, recovery and persistence. It will also include the concept of groups where a list of receivers belong to a group. A message sent to the group will be delivered to all the addresses registered in that local group. The second model will provide for asynchronous and deferred synchronous communication without recovery.

Both implementations are designed to take advantage of OODCE-provided DCE-Pthread class which is used to start and control the execution of a thread. The second mode requires more programmer involvement than the first model. Message Passing Service is generally intended to handle low volumes of data per message. Compare with k/ftp (below) for bulk data transfer.

Thread

A thread is a light weight process without the actual process overhead. Threads provide an efficient and portable way to provide asynchronous and concurrent processing, which is a requirement of network software. Threads can maintain thread specific data and can also share data with other threads in an application. This service provides functionality to create, maintain (scheduling, locking, etc.) threads.

Time

The Time Service keeps system (host) clocks in the ECS network approximately in sync by adjusting the time kept by the operating system at every host. This service changes the clock tick increments (rather than the actual clock) so that host clocks will be in sync with the some reference time provided by an external time provider. CSS will also provide a way to simulate time by applying a supplied delta time to the actual time. With in ECS, OSF DTS will be used to sync the system clocks. For more information please refer to Section 3.4.

LifeCycle

Managing a system involves managing individual applications. An operator may want to start a new application, shutdown/suspend a running appellation due to anomalies. An application may not be active all the time to accept requests. In order to effectively use the CPU and memory it is desired to control the applications as well s some objects residing in the application by starting them on demand.

LifeCycle services can be broadly classified into two categories: Application and Object level. LifeCycle services for applications involve Startup, Shutdown, Suspend and Resume functionality on applications. This functionality lets the M&O manage server applications. MSS provides the application related LifeCycle functionality. CSS provides the internal APIs that are needed for the MSS to control the applications. LifeCycle services for objects provide the application programmer with the functionality to create and delete server objects residing in different address spaces.

Distributed Object Framework (DOF)

In an object oriented processing architecture, objects may be distributed in multiple address spaces, spanning heterogeneous platforms. The basic contract between an object and its users is the interface that the object provides and users can use. Objects can be spread across the network for reasons of efficiency, availability of data, etc. From the perspective of the requester of a service, invocation should be the same no matter where the object physically resides.

The distributed object framework will be implemented using OODCE. The set of core DCE services are naming, security, threads, time, rpc. In order to aid the application programmer, another layer of abstractions is provided with OODCE. Four generic classes: DCEObj, DCEInterface, DCEInterfaceMgr, and ESO will be available for application programmers to implement client-server applications.

Electronic Mail (E-Mail)

E-mail is a standard component of Internet systems. It is useful for asynchronous, relatively slow notification of many different types. Also, E-mail is persistent, and will continue to try to deliver even if there are temporary network outages. The CsEmMailRelA class provides object-oriented application program interface (API) to create and send e-mail messages.

File Access-k/ftp

FTP is a Internet standard application for file transfers. It allows a user to retrieve or send files from/to a remote server. The files transferred can be either ASCII or binary files. FTP also provides an insecure password protection scheme for authentication. KFTP builds on the standard FTP but adds a layer for strong Kerberos authentication. The CsFtFTPRelA object provides an object for managing FTP sessions between clients and servers to allow programmers to transfer files between machines.

Bulletin Board

Bulletin Board is another Internet standard application, however unlike e-mail, Bulletin Board messages are directed to all readers of a named group. It uses the Network News Transfer protocol (NNTP) for sending and receiving messages. The CsBBMailRelA object provides object-oriented application program interface to send e-mail.

Virtual Terminal

Virtual Terminal access refers to remote log on to a machine. This software is provided on all platforms. The server telnetd will listen to incoming telnet clients and will allow remote logons. There is also a secure version of telnet and telnetd using Kerberos authentication which CSS will provide where available. This service is allowed only within ECS due to security considerations.

X is a Graphic User Interface conforming to the X/Open standard. While X is not a specific CSS Release A service this description is listed here for informational purposes. It consists of a client and a server where the client displays the actual interface. Developing applications in X is cumbersome and complex. OSF Motif is another standard, layered on top of X which provides a high level application programming interface to make the application development easier. Applications developed with Motif will work with an X server. The X client/server connection presents some significant security risks; therefore ECS will not support applications where the X

client and the X server reside on different platforms. Users can down load data from ECS and can use the X application to view the data on their local machines. Alternatively, the program should consider providing dedicated circuit access from a user client to connect to an ECS X application.

Event Log

Event log provides the programmers the capability to record events in to files. Events are broadly classified into two categories: management events and application events. Each event is recorded with all the relevant information for identifying and for later processing. Management events needs to be recorded in a history file and on some occasions reported to the Network Node Manager. Application events are only recorded into a programmer specified file. Event log provides a uniform way for the application programmers to generate and report (record) events.

3.1.2 CSS Hardware

The CSS hardware at the EOC consists of the following components:

- Communications Servers
- Management Workstations
- Printers

3.1.1.1 Connectivity

The EOC hardware connectivity is depicted in Figure 3.1.2-1. The EOC LSM resides on a separate FDDI ring, with connectivity to FOS systems and the rest of the site provided by a redundant FDDI switch/router. The MSS Local Management Server is equipped with a DAS (dual-attached station) card that is connected to two FDDI concentrators, providing redundancy in the event of a concentrator failure. The MSS monitoring workstations are equipped with an SAS (single-attached station) interface card, connected to a single FDDI concentrator.

3.1.2.2 CSS Hardware Components

The CSS communications server is the primary server for CSS applications and data. It is cross-strapped to the MSS monitoring / management server to provide for failover (warm standby) capability, and is populated with CSS applications and as well as the OSF Distributed Computing Environment (DCE) software (e.g., DCE client services, DCE Directory Server, DCE Security Server, DCE Time Server),

The management workstation configurations are networked workstations that support all aspects of enterprise management between the M&O staff and the LSM. The management workstations are populated with the CSS client, the MSS management agent services, and user-selected subsets of the enterprise monitoring configuration software and data.

3.1.2.3 Failover and Recovery strategy

Analysis of failover strategies supports the integration of the CSS and MSS servers to serve as warm standby to each other, cross-strapped to RAID devices for critical data access by either server. CSS logical server functions are be configured but inactive on the MSS server. In the event of a failure of either server, the second RAID can be mounted for use by the backup server. All data is replicated, and is also routinely safestored in the ECS data server archive.

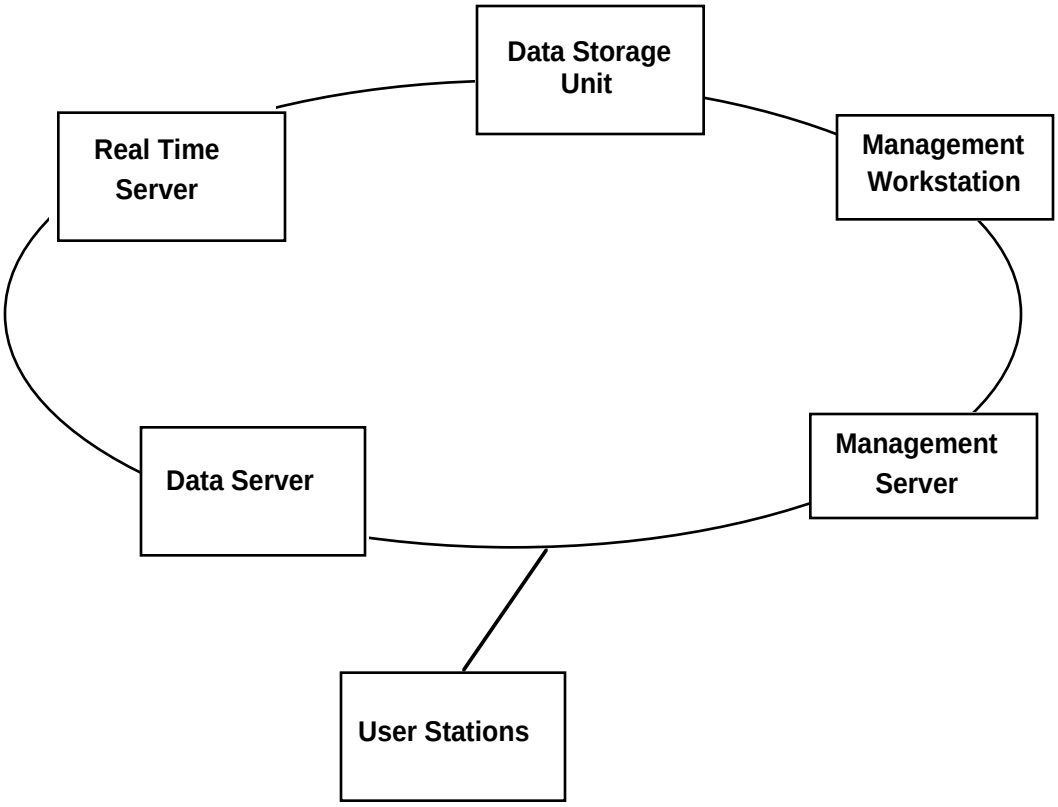

Figure 3.1.2-1. EOC Hardware Connectivity

Specific calculations of reliability and availability of CSS components are provided in 516-CD-001-003, Reliability Predictions for the ECS Project (August 1995), and 518-CD-001-003, Maintainability Predictions for the ECS Project (August 1995).

3.1.2.4 Security Implementation Overview

The CSS, MSS and ISS subsystems all contribute to the ECS security implementation. ISS provides router-based security based on TCP/IP packed [IP address and port] address filtering. CSS provides Kerberos-based authentication, integrated authorization based on DCE access control, and data integrity based on encrypted checksums (provided by DCE). MSS provides virus checking, accountability, event reporting and analysis, and security policy coordination. Table 3.1.2.4-1 presents ECS security components and their contributions to overall security requirements. The CSS addresses the first four needs.

Table 3.1.2.4-1. Components of CSMS Security Implementation

Security Need	CSMS Security Implementation
Authentication	DCE-based Kerberos. Kerberized ftp, Kerberized telnet.
Authorization and access control	DCE access control. Router-based filters (port/socket at transport layer, and source and/or destination address at network layer). DCE cell configuration / "iso-cell" partitioning.
Data integrity	DCE-based RPCs (encrypted checksums).
Data confidentiality	DCE-based RPCs (encrypted data).
Countermeasures for degradation in network or processing resource performance through denial of service attack	Router-based filters; Support for nsu-IP-routing, application-layer gateways and bastions.
Security database management	DCE ACL managers, registry database.
Compliance management	MSS COTS & public domain tools for password audits, file system integrity checking.
Intrusion detection	COTS for detecting viruses, worms, Trojan horses, public domain tools (e.g., TCP Wrapper).
Security reporting	RDBMS.

In addition to service-based contributions to security implementation, ECS has an integrated logical and physical security partitioning strategy involving DCE cell partitioning (Cell partitioning is a Release B feature) and use of isolation LANs at each of the ECS sites. Thus, although CSS encapsulates DCE services into higher-level, more abstract services for application developers, DCE plays an important role in maintaining the integrity of the entire ECS.

3.2 Interprocess Communication (IPC) and Notification/Callback Services

Interprocess Communication Service (IPC) is provided in several forms for FOS subsystems to use. In this section, the CSS provided IPC services and their interfaces are presented. Section 3.1.1 presents the Interprocess Communication/HCL interfaces. Section 3.1.2 presents the Message Passing Service interfaces for synchronous, asynchronous and deferred synchronous communications. Section 3.1.3 presents the Multicasting Service interfaces.

3.2.1 Interprocess Communication/HCL Overview

In order for FOS processes to communicate with each other, an interface called EcMpTransport was defined to give global access to certain capabilities of the transport layer and protocol for specific implementations of the transport layer. Each underlying implementation must provide a derivation of the EcMpTransport class that the user instantiates to do interprocess communication and notification.

For the HCL implementation, there are two derivations: EcMpHCLMotifTransport for processes which are Motif based, and EcMpHCLUnixTransport for processes which are non-Motif based.

In addition to the EcMpTransport class, five other classes are provided to support message passing. The EcMpAddress class describes the address of FOS services running somewhere in the system. EcMpBindingInfo represents an abstract class for information needed to make a connection to a specific address. EcMpMessageCb is an abstract ecs message callback class. EcMpMessageIf provides services to send and receive ecs messages or to send receive buffers of binary data; it is derived from RWCollectable. The EcMpNameServer is an abstract base class providing global access to the name server and protocol for specific implementations of the name server.

3.2.1.2 Object Model

Table 3.2.1.2-1 summarizes the Interprocess Communication and Notification classes.

Table 3.2.1.2-1. Interprocess Communication and Notification Object Responsibility Matrix (1 of 2)

Class Name	Description
EcMpTransport	Abstract base class providing global access to transport layer and protocol for specific implementations of the transport layer.
EcMpAddress and EcMpBindingInfo	EcMpAddress contains address information for identifying a specific instance of an EcMpMessageIf object residing somewhere on the network. The member data of EcMpAddress was specified by FOS as the data they would like to have to identify interfaces. It follows: Information — Description — Default/Wildcard processType — "?" processMode — "?" processName — "?" group — "?" host — name of host — "?" user — name of user — "?" role — "?" spacecraft — "?" mode — "?" In addition, there is a myObjRef member data that is set by the EcMpMessageIf to which the address is assigned to in order to make sure that the address is unique. There is also a pointer to some derivation of EcMpBindingInfo which is also set by the EcMpMessageIf to which the address is assigned. For HCL, this is an instance of EcMpHCLBindingInfo, which contains a host name and port number determined from the host and port of the process which contains the EcMpMessageIf instance. All of this information is used in the name service entry identifying the EcMpMessageIf instance

Table 3.2.1.2-1. Interprocess Communication and Notification Object Responsibility Matrix (2 of 2)

Class Name	Description
EcMpNameServer	EcMpNameServer is an abstract class providing global access to the name server and protocol for specific implementations of the name server. Each underlying implementation must provide a derivation of the EcMpNameServer class that can be used to access the name service. This particular EcMpNameServer knows the details of the specific EcMpBindingInfo which goes with the implementation so it can make the appropriate entries in the DCE name service. For HCL, this derivation is the EcMpHCLNameServer class. Note that when you call EcMpTransport::init(), the appropriate instance of EcMpNameServer is automatically set up and made available for you to use. The EcMpNameServer is used internally by EcMpMessageIf when you ask it to add its address to the name service, thus making it possible for other interfaces to find it on the network.
EcMpMessageIf and EcMpMessageCb	EcMpMessageIf is an abstract class specifying the protocol for a message interface object which enables you to send and receive messages and data. Each underlying implementation provides its own derivation of EcMpMessageIf that implements these capabilities according to its own interprocess communication scheme. For HCL, this is the EcMpHCLMessageIf class. When you call EcMpTransport::createNewIf(address), you automatically get an instance of the right kind of EcMpMessageIf. You should think of each EcMpMessageIf as an object that exists somewhere on the network. This object has a unique EcMpAddress which you give it specifying how you want it identified (i.e. processType, processName, etc.) The EcMpMessageIf itself adds a unique object reference and its binding information. You can use this object to establish communication with some other object on the network by calling the estConn() member function with the address of the object to which you want to communicate. Once this is done, when you call the send() or sendAndWait() behavior, the message or data is sent to the other object. By calling the addAddressToNs() behavior, you cause the EcMpMessageIf object to publish its address on the name service, making it possible for other EcMpMessageIf objects to find it and send it messages or data. You process this data by giving the EcMpMessageIf object a derived instance of an EcMpMessageCb object which you create. When an EcMpMessageIf object receives a message or data directed at it from another EcpMessageIf object, it calls the handleMessage() or handleData() member function of your derived EcMpMessageCb object which presumably knows how to process the data. If you want to send a reply to the sender of the message or data, you call the reply() member functions of your EcMpMessageIf object which received the message or data. Note that a "message" is an instance of a RWCollectable object. (FOS has prototyped an EcRWCollectable class derived from RWCollectable that will become the message.) "Data" is just an unsigned char buffer of some length that gets sent without any knowledge of what is inside.

3.2.1.3 Sample Application Programmer Interface

The following example shows how EcMpTransport will typically be used.

```
int main(int argc, char **argv)
{
  ....
        // Create an instance of the transport layer and make its global access available
        EcMpHCLUnixTransport myTransport;
        EcMpTransport::setTransport( &myTransport);

        // Ask transport layer to initialize.
        // For HCL, this means doing things like initializing the notifier and message
        // agent.    Note that the init() will automatically initialize an instance of
        // EcMpNameServer,  making it available for interfacing with the name server.
        // If the developer doesn't need the name server,  the initIpcOnly() member
        // function can be called.
        if( !EcMpTransport::init())
                {
                 ... something wrong
                return 1;
        }

        .... do some stuff, like maybe set up some instances
        .... of EcMpMessageIf (more on this later)

        // Now start polling for automatic notification when events happen.
        if (  !EcMpTransport::startPoll() )
        {
        .... something wrong
        return 1;
        }

        // The notifier was stopped, maybe by a call to stopPoll() somewhere, so just exit
        // the program
        return 0;
        }
```

If automatic polling is not desired, and the developer wants to manage his/her own event loop,then the EcMpTransport::checkForActivity() and EcMpTransport::waitForActivity() can be used instead of startPoll().

The following code fragments show how EcMpAddress, EcMpNameServer, EcMpMessageIf, and EcMpMessageCb will typically be used.

```
Set up an address for a network message interface object.
EcMpAddress *adr = new EcMpAddress;

adr->processType("sc_control");
adr->processMode("primary");
adr->processName("bob");
.... fill out rest of address

// Create the object assigning it the above address
EcMpMessageIf *msgIfObj = EcMpTransport::createNewIf( adr);

// Instantiate a callback for handling messages directed at the object
MyCallBackClass *cb = new MyCallBackClass;
// NOTE:  you derive MyCallBackClass from EcMpMessageCb
msgIfObj->messageCb( cb);

// Publish the object's address with the name service so other objects can find it to
// send messages and data
msgIfObj->addAddressToNs();

// Now if the developer wants to use this object to send messages and data to
// another object on the network, all that is needed is that this object be of type
// sc_display and of  primary mode.
// Use a mask address in a call to the name server to locate all registered objects
// which match the criteria.

EcMpAddress maskAdr;
maskAdr.processType("sc_display");
maskAdr.processMode("primary");
```

```
EcMpAddressList possibleAdrs;
if( EcMpNameServer::findAddress( possibleAdrs, maskAdr))
{
// Just pull the first address out of the list
EcMpAddress *rcvr = possibleAdrs.first();

if ( msgIfObj->estConn( rcvr) )
{
MyMsg msg;  // NOTE:  MyMsg is derived from RWCollectable (or
                // EcRWCollectable)
msgIfObj->send( &msg);
}

possibleAdrs.clearAndDestroy();
}

// NOTE that instead of going to the name server implicitly,  the developer could
// have just called msgIfObj->estConn() with the mask address as a parameter, and
// it would have gone to the name service automatically to locate a complete
// address satisfying the mask criteria.  The above example was just done to
// illustrate name service usage.
```

Now suppose that a message directed at msgIfObj comes in. Then the handleMessage() member function of MyCallBackClass will automatically be called from within either the EcMpTransport::startPoll() method if doing automatic polling, or the EcMpTransport::checkForActivity() or EcMpTransport::waitForActivity() methods if doing my own polling.

```
MyCallBackClass :: handleMessage(RWCollectable *msg,
                                 EcMpMessageIf *msgIf,
                          EcMpAddress   *senderAdr)
{
.... do some processing on the message
// now I want to send a response
MyRspMsg msg;  // derived from RWCollectable or EcRWCollectable
```

```
msgIf->reply( &msg);
....
}
```

The above code fragments do not show all the behaviors of these classes, but they do illustrate some typical usage.

3.2.2 Message Passing Overview

ECS distributed computing consists of a variety of client and server applications running on different platforms. Clients send data to servers, which process the data and return the result to the client. This interaction can be classified into three categories: synchronous, asynchronous and deferred synchronous.

In synchronous mode, a client makes a request and passes control to the server, i.e., waits for an acknowledgment. The server services the request and returns the result back to the client, at which point the client gets back the control. The program execution on the client side is blocked until the server returns from the service. This is a blocking call and is referred as "synchronous".

In asynchronous mode, the client makes a non blocking request. Client processing can continue simultaneously with the server processing. Asynchronous message passing can be further subdivided into two parts: guaranteed and non-guaranteed. Guaranteed asynchronous message passing guarantees the delivery of the message to the receiver (client-to-server or server-to-client). The sender of a message can verify the delivery of the message through acknowledgments. In non-guaranteed asynchronous message passing, the sender should send the message to the receiver only if the receiver is active and listening. The receiver still may not receive the message due to any number of reasons, and this is not considered an error. FOS applications use this method to send real time telemetry data to ISTs asynchronously.

Deferred synchronous model is a superset of the asynchronous model. In deferred synchronous mode the client makes a call and receives an acknowledgment of the call, but no results; it also gets control back right away. Once the receiver completes doing any processing with the data, results are sent back to the client.

Table 3.1.2-1 describes the different types of message passing and when they are generally used.

CSS will implement synchronous, deferred synchronous, and guaranteed asynchronous message passing using Distributed Object Framework. Deferred synchronous communications involve some degree of application programmer involvement.

FOS applications like the Off-line Analysis Request process uses the message passing service to send analysis data to the Off-line Analysis process and to receive the results of such analysis.

The FOS ECS Operations Control uses the message passing service to send schedule information to the ISTs.

SDPS process-intensive applications send the intermediate processing state/results to User Interface (UI) to display the results of the process done so far.

SDPS subscription service (between the science DataServer and the product generation) needs guaranteed asynchronous message passing.

CSS is providing two implementations for Message Passing. Both are designed to work with the OODCE-provided DCE-Pthread class which is used to start and control the execution of a thread.

The first implementation provides asynchronous and synchronous communications (byte streams only) with store and forward, and recovery - persistence. The concept of groups (one or more receivers) is included into this API.

The second implementation provides asynchronous and deferred synchronous communication. The CSS API for this design is not going to be as transparent to the user as the CORBA application will be, that is, some developer involvement will be required. With respect to Store and Forward, a thread can be created so that it can periodically try to send the message. The CSS API can handle any data types; recovery will not be supported.

Table 3.2.2-1. Message Passing Communication Types Defined (1 of 2)

Type	Description
Synchronous	Normally, this is how clients and servers interact using the Remote Procedure Call (RPC) mechanism. Both the client and server have to support the same interface. Servers are continuously listening for incoming requests. Each server, while supporting the same interface, can implement the interface differently. The client has the choice to bind to a server offering a particular implementation of the given interface. The type of arguments that can be used in this mechanism are all the system defined generic types plus user defined types. This is a blocking call, where the client has to wait until control returns back to it from the server.
Non Guaranteed Asynchronous	This is same as the synchronous message passing except that this is non-blocking, i.e. the client gets the control back immediately without waiting for the server processing to complete. In this mechanism, no result is returned. When a client invokes the request, the system invokes all the servers currently listening that support the interface. This is used primarily when the client wants to send some information to interested parties and doesn't care whether they receive it not. For example, when the CDS server comes up, it sends a message of its existence to all the (internal framework) directory agents that are listening.
Guaranteed Asynchronous	In guaranteed asynchronous mode, the sender specifies a list of receivers where he/she wants to send the message. A UUID is received back when the send request is issued. When a callback is returned indicating that the send operation completed, a UUID is passed back identifying which message was sent successfully or not. This is also a non blocking call, returning control to the sender immediately. The delivery of message is guaranteed in this mode. If a receiver is not listening, the message will be buffered and sent to the receiver at a latter time. The argument types that can be passed are not as general as in the above two cases, rather restricted to string type. This is used where large quantity of data needs to be passed, without any processing, e.g., the FOS Product and Scheduling element passing an instrument schedule to the SCFs.

Table 3.2.2-1. Message Passing Communication Types Defined (2 of 2)

Type	Description
Deferred synchronous	In deferred synchronous message, a sender sends a message to one receiver and gets back a UUID. This UUID is then used at a latter time to receive the result back from the receiver. This is used in process intensive applications, where the receiver takes some time to process the request. While the processing continues at the receiver, control is returned back to the sender immediately. As in guaranteed asynchronous message passing, the argument types are restricted to strings only.

3.2.2.1 Message Passing Implementation One

3.2.2.1.1 Overview

The first implementation will provide for asynchronous and synchronous message passing with store and forward, and recovery - persistence. The concept of "groups" is included in this API. A group is a symbolic name representing a number of receivers. Each receiver will be identified by a logical name associated with a UUID. This API will handle messages consisting of only byte streams.

3.2.2.1.2 Context

Message Passing is an infrastructure key mechanism and is used by ECS subsystems for synchronous, and asynchronous communication where a client needs control back immediately after invoking a remote procedure.

CSS will provide a set of classes (custom code) that will allow FOS and SCDO subsystems to achieve synchronous, and asynchronous communication. Security is available on this architecture.

3.2.2.1.3 Directives and Guidelines

Directives:

The programmer must do the following in order to transfer data:

- Define a new class inheriting from the generic EcDcDSyncCom and must implement all the virtual functions defined in the parent class, PreInvoke, Invoke, and PostInvoke.
- Include the ClidDSyncCom.h header file.
- Create a client or a list of clients.
- Set the data pointer to some data that will be used in the overridden Invoke member function.
- Set the address pointer. The address pointer can be an IP, a port number, an object reference, a CDS name, etc.
- Optionally, set the thread policy attributes or just go with the defaults (EcDDcPri_min, and EcDDcFg)
- Optionally, set the number of retries in case of exceptions/errors. Defaults can be used.
- Optionally, set the time between retries in seconds. Defaults can be used.

- Call the method Send to transfer data.
- Call Done to find out whether the thread has finished execution successfully.
- Call Reset to set the flags to initial values and deallocate memory.
- Either re-send or terminate the session.

Guidelines:

This Message Passing implementation utilizes COTS such as OODCE and RogueWave. For information on OODCE please refer to the HP Object Oriented DCE C++ Class Library Programmer's Guide. For information on Rogue Wave please refer to SunSoft C++ 4.0.1 Tools.h Class Library.

It supports persistance. Intermediate queues are maintained, which collects all the messages to be sent, along with the receiver information and then sends them to the intended receivers. When a message is to be sent asynchronously, a pointer to it is placed in to an outgoing queue and control returns right away. The contents of the queue are saved to disk. If no file name is supplied when instantiating the control call, recovery/persistence is not provided.

Worker threads operate on the outgoing queue and will process simultaneous send operations. Each request that arrives is placed at the end of the outgoing queue. After adding the request to the queue, the boss thread will wake up a worker thread and this worker thread will perform the send operation. Once done, it will wait for the next request. If the message fails to be sent, it is placed back into the queue, the 'time_sent' field is updated, and, it is retried when it reaches the specified retry time. If it fails all the tries, the message is returned (callback is invoked with its UUID, destination, message, and send status).

Priorities are assigned to the spawned threads. Initially CSS is designed to support ten threads, with a configurable number of threads at each priority. A total of five priorities are defined. A higher priority thread tries to find a message from its corresponding queue and then sends it. In the absence of any messages in its queue, it looks for messages in the lower priority queues to send. Threads are assigned for each priority (even the lowest priority). This is done to ensure that lower priority messages are not starved.

In asynchronous mode, when the client sends the data to a set of receiving processes, a pointer to the data is kept in intermediate message queues and control comes back to the caller immediately. The message passing logic will locate the server who is continuously listening, and sends the data to the server.

The server can be running continuously to receive requests from the client. Alternatively, if the server is not running when the message is sent, the message passing logic can periodically try to send the message. The sender can specify the number of send-tries, and the time between tries. After sending the data, the sender will receive an acknowledgment indicating success/failure upon completion of the send operation.

In this mechanism, each application can receive as well as send messages to other applications. Each application maintains a group of outgoing queues in which outgoing messages are kept along with other information like destination. Each outgoing queue is associated with a priority. Messages belonging to a particular priority are kept in the corresponding outgoing queue. At startup, several threads are spawned whose primary purpose is to pop messages from the outgoing

queues and send them to the proper destination. Each message is also associated with the number of tries, that is, how often it should be sent before declaring a failure (due to unavailability of the receiver or network failures) and the time period between each retry. The threads try to send the message the given number of times and calls an application supplied call back either after a successful send or after the determination of a failure to send the message.

A distributed object is defined with the functionality to transfer messages. After selecting a message from a queue, a thread creates a distributed client (a proxy object) and invokes the transfer method to transfer the message to the server object (the receiving application).

In order to receive messages, each application creates a receiving queue with a unique name. An application can create as many receiving queues as needed, each with unique name. This unique name is used by an application to send messages and it is kept in the Cell Directory Name Space (CDS) along with the binding information.

Each receiving queue is associated with a distributed server (a transfer object), which can receive messages from other applications. When a message is received by the server object, that message is kept in the queue associated with that object. The ordinary receiving queue provides two methods: read and read wait. These methods allow the application programmer to retrieve messages from the queue. The 'GetMessage' method performs a single read operation and returns a message from the incoming queue or Null (there are no messages in the queue) and a return status. The 'GetMessageWait' method performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call, or a default time period will be used.

Additionally a call back incoming queue is provided. This queue contains one thread which executes a programmer-defined call back every time a message is received. That is, a thread is awakened the moment a message arrives in the incoming queue, it invokes the virtual method 'handleMsg', and searches to see if any other messages arrived, if so, it executes another call back, else, it shuts down the thread until a new message is received. When a message is read, it is removed from the queue and from the disk.

Deferred synchronous model is a direct superset of the asynchronous model. While implementation one does not support deferred synchronous directly, it provides certain information, such as, the message UUID, so the application programmer can implement deferred synchronous messages in the application. Alternatively, the application programmer can use the CSS implementation two to provide deferred synchronous message passing as an infrastructure service.

Constraints:

- Each queue must have a maximum number of messages, a user-defined number.
- The messages must be of defined maximum length.
- Logical names ought to be unique, internally CDS/GDS will be used to store location information associated the receiver (logical name-UUID pair).
- The message or buffer should not be changed while the sending is in progress; otherwise, we need to copy the message.
- If deferred synchronous is needed, a UUID and the sender's logical name can be used to send a reply back by the receiving end.

- The Message or Buffer should not be changed while the sending is in progress. Otherwise, we need to copy the message.

3.2.2.1.5 Sample Application Programmer Interface

Sample #1

This scenario demonstrates how a process can be a sender of asynchronous messages and a receiver as well. It also demonstrates the usage of a receiver callback to receive a message (A process sends a message to another process and is able to receive as well).

Description

a. Setup an ordinary receiver session

1. Instantiate an object of the EcMpMsgPsngCtrl type using a filename (for the outgoing queue and for the disk filename/persistence), and an application name (a full CDS path name + application name).
2. Using the message passing control object start listening for incoming messages (Initialize). Next, instantiate receiving objects, that is, create an instance of the incoming queue object (via the 'createReceiver' call) and set it to point to the queue the message passing control class is attached to it.
3. When a message comes in, it gets stored in to the queue associated with the object and a copy of it is stored on the file associated with the object.
4. In order retrieve a message two methods can be used: 1) GetMessageWait (block - there is wait) Performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call; 2) GetMessage (Non Block - there is no wait) Performs a single read operation and returns a message from the Incoming Queue or a return status (“no more messages”).

b. Setup the sending session

1. A sending session is basically a point-to-point session to send a message. Each sender session will have a logical name that is needed to contact the receiver. A list of sending sessions are maintained in a given application. A sender session list contains a callback object which provides virtual functions to be called when a send is complete. This is done at the sender side.
2. Create a callback object and implement/compile the acknowledgment virtual function. The object EcMpMsgCb has a virtual function named 'handleAck', this function is called when a message is delivered to the destination or when the underlying mechanism failed to deliver it within the given constrains(number of tries). This function takes the following arguments: a flag indicating success/failure, the number of tries, the time between tries, the actual message, its length, a receiver address, and a destination address, and a message identifier. This same unique id is presented to the application programmer when sending a message.
3. Create a EcMpSessionList and populate it with the following information: 1) Associate the callback object just created, 2) Set the number of tries and the time between each retry, and 3) For each session give the receiver’s (destination) logical name.

Prepare and send the message to the session list. At initialization time a number of threads were generated internally which periodically will get the messages from the outgoing queue and send them. The send call returns back a UUID so the sender that can be used by the callbacks. This UUID can also be used to get the reply back if there is one.

c. Setup a callback receiver session

1. The receiver will be notified everytime a message is received. A thread is awakened when a message is received in the queue. A callback is invoked at that point.
2. Instantiate an object of the EcMpMsgPsngCtrl type using a filename (for the outgoing queue and for the disk filename/persistence), and an application name (a full CDS path name + application name).
3. Using the message passing control object start listening for incoming messages. Next, instantiate receiving objects, that is, create an instance of the callback incoming queue object (via the createReceiverCb call) and set it to point to the queue the message passing control class is attached to it.
4. When a message comes in, it gets stored in to the queue associated with the object and a copy of it is stored on the file associated with the object. A 'call back' thread (just one thread) will be awakened every time a message arrives into the special receiver queue created by the call 'createReceiverCb'. A call back (virtual function) will be executed. It will check if there is another message and if so execute another callback. If there are no new messages it shutdowns the thread until a new message arrives in.

Step 1 Initialize, and **Step 2** Setup Receiver Sessions

/*

The application programmer should include a set of include files given by CSS at the start of the application.

The EcMpMsgPsngCtrl object is the controller object, through which any number of receiver sessions can be created. Each receiver session is associated with a unique name (so other applications can send messages to this receiver) and a unique file. This file is used for persistence. When a message comes in, it is stored in to the queue associated with the object and a copy of it is stored on the file associated with this object.

Internally, this object also creates a sender queue. All outgoing messages are kept in this queue.

A number of threads are generated internally in the initialize call which periodically get messages from the outgoing queue and send them

*/

```
// Instantiate the controller object (initialize)
Extern EcMpMsgPsngCtrl* theMsgCtrlP;
theMsgCtrlP  = new EcMpMsgPsngCtrl(fileName1), //to store outgoing messages
                                        applName1); //CDS Name - applic. name
// Start listening
EcMpMsgPsngCtrl->Listen():
```

```
// Instantiate receiving objects.
EcMpQueueIn   *rec1P;
EcMpQueueCbIn *rec2P;

// Create an ordinary receiver
// Open a disk file 'fileName2' for this receiver queue (for
// persistence purposes)
rec1P = theMsgCtrlP->CreateReceiver(receiverUniqueName,
                                    fileName2);

// Instantiate the receiver callback object
EcMpMsgCb* msgRecCB_P = new EcMpMsgCb();

// Create a special receiver
// This receiver executes a virtual callback function
// the moment a message is received.
// Also, open a disk file 'fileName3' for this receiver queue
// (for persistence purposes)
rec2P = theMsgCtrlP->CreateReceiverCb(receiverUniqueName,
                                      fileName3,
                                      msgRecCB_P);
```

Step 3 Setup the Sending Sessions

/*

A sending session is basically a point to point session to send a message. Each sender session will have a logical name that is needed to contact the receiver. A list of sending sessions are maintained in a given application. A sender session list contains a callback object which provides virtual functions to be called when an asynchronous send is complete. This is done at the sender side.

Create a callback object and implement (compile) the acknowledgement virtual function. The object EcMpMsgCb has a virtual function named handleAck. This function is called when a message is delivered to the destination or when the underlying mechanism failed to deliver it within the given constrains(number of tries). This function takes the following arguments: a flag indicating success/failure, number of tries, time between tries, the actual message, destination, a unique id. This same unique id is presented to the application programmer when sending a message.

*/

```
// Instantiate the sender callback object
EcMpMsgCb* msgSenderCB_P = new EcMpMsgCb();

/*
Create a sender session list and populate it with the following information:
    1. Associate the callback object.
    2. Set the number of tries and the time between each retry
    3. For each session give the receiver (destination) logical name.
In case of a synchronous send method, the callback is not used. As such a null argument is a valid
one in the constructor. Similarly number of retries and time between calls are applicable only for
asynchronous sends.
 */

// Instantiate sessions lists and give information (callback, # of tries, destination names ...)
EcMpSenderSessionList *ssListP=new EcMpSenderSessionList(msgSenderCB_P);
ssListP->SetTries (5,10);  // no of tries, and time between tries
ssListP->Join(LogicalName); // logical name (stored internally in CDS with its
                                        // respective object UUID
ssListP->Join (anotherLogicalName);
```

Step 4 Send a message

```
/*
Send a Message: This returns (output argument) an UUID so the sender can use it in the  callbacks
and the status of the call.
*/
// Send the message to the group (all in the group will receive it)
status<- ssListP->Send(WaitFlag,  // Wait (Synch) or  Non-Wait (Asynch) Flag
        msgP,          // A void pointer
                msgLength,      // Msg length
                senderAddr,     // Destination queue
                recAddr,        // Client Queue. If NULL, no receiving at the client
                                        // will occur
        priority,       // Priority 1/2/3/4/5
                uuidFlag,       // UUID will be given or not
        msgId);         // This is an output argument to identify the message
                                        // in the callback method
```

/*

In case of a synchronous call, control returns after the call is completed. In this case, the callbacks are not called. Callers main thread is used to send the message. In case of asynchronous calls, the message is put in a queue, which will be sent later by an internal thread.

*/

Step 5 Receiving a message

/*

a. Read - Block (there is wait): Performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call.

*/

```
rec1P->GetMessageWait(msg);// wait and dequeue
```

/*

b. Read - Non Block (there is no wait)

Performs a single read operation and returns a message from the Incoming Queue or a return status ("no more messages")

*/

```
rec2P->GetMessage(msg);// dequeue
// ... anything else you want to do ...
```

3.2.2.1.6 Object Model

Table 3.2.2.1.6-1 summarizes the first Message Passing Service classes which are discussed in detail in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project.

Table 3.2.2.1.6-1. Message Passing Object Responsibility Matrix (1 of 2)

Class Name	Description
EcMpMsgPsngCtrl	The EcMpMsgPsngCtrl object is the controller object, through which any number of receiver sessions can be created. Each receiver session is associated with a unique name (so other applications can send messages to this receiver) and an optional unique file. The file is used for persistence. When a message comes in, it is stored in the queue associated with the object and a copy of it is stored on the file associated with this object. Internally, this object also creates a sender queue. All outgoing messages are kept in this queue. A number of threads are generated internally in the initializing call which periodically get messages from the outgoing queue and send them.
EcMpMsgCb	This class will handle two types of callbacks: 1. For ordinary receive messages: If an ordinary message is received, then handleMsg is invoked; 2. For acknowledgment of messages: If an acknowledgment is received, then handleAck is invoked. A sending session is basically a point to point session to send a message. Each sender session will have a logical name that is needed to contact the receiver. A list of sending sessions are maintained in a given application. A sender session list contains a callback object which provides virtual functions to be called when a send is complete. This is done at the sender side. A callback object is created and will implement the acknowledgment. The virtual function handleAck is called when a message is delivered to the destination or when the underlying mechanism failed to deliver it within the given constraints (number of tries).
EcMpSessionList	This is a container class whose element type is a logical name and will inherit from the RWTPtrSlist class. A session list contains a callback object which provides virtual functions to be called when a send is complete. This is done at the sender side.
EcMpQueue	This class will be the parent class for the following: . EcMpQueueIn . EcMpQueueCbIn . EcMpQueueOut EcMpQueue inherits from the Rogue Wave library file RWPtrDlist.
EcMpQueueCbIn	This queue will contain one thread which will execute a callback every time a message is received. The callback will be a virtual function call. This class defines a double linked list queue. It inherits from the Rogue Wave Library file, RWTPtrDlist.

Table 3.2.2.1.6-1. Message Passing Object Responsibility Matrix (2 of 2)

Class Name	Description
EcMpQueueIn	This class will be used to queue the messages once they are received. It will provide a Read Wait call and a Read Non-Wait call. The Read Wait call performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call, or a default time will be used. The Read Non-Wait performs a single read operation and returns a message from the Incoming Queue (or Null if there are no messages in the queue) and a return status. This class defines a double linked list queue. It inherits from the Rogue Wave Library file, RWTPtrDlist.
EcMpQueueOut	In case of a asynchronous calls, the message is put in a queue, which will be sent later by an internal thread. Controls returns immediately. Worker threads will process simultaneous send operations. Each request that arrives is placed at the end of the outgoing queue. After adding the request to the queue, the boss thread will wake up a worker thread and this worker thread will perform the send operation. The send operation will not remove the item from the queue yet. Once done, it will wait for the next request. There will be about five to ten working threads and one boss thread. If the message fails to be sent, then the 'noOfTries' gets decreased by one, and the 'lastTimeSent' gets updated to the current time (when the message came back after the send failed). The message will be retried once the 'lastTimeSent'+'timeBetweenTries' was reached until the noOfTries expired. If the message failed to be sent, the message will be returned (the callback 'handleAck' will be invoked). The EcMpQueueOut class defines a double linked list queue. It inherits from the Rogue Wave Library file, RWTPtrDlist.
EcMpTransferSrv	This class is the EcMpTransferSrv manager object. It responds to client's requests to transfer data which result in the enqueuing of the data.
EcMpTransferCli	Class EcMpTransferCli is a surrogate object for making requests to an EcMpTransfer manager object. An EcMpTransferCli object creates and holds a single instance of this class which it then binds to successive EcMpTransfer manager objects to carry the transfer operation.

3.2.2.1.7 Dynamic Model Scenarios

3.2.2.1.7.1 Scenario #1

Abstract

- This scenario will demonstrate how a process can be a sender of asynchronous messages and a receiver as well.

Interfaces

- OODCE provided classes.

Stimulus

- The process wants to send a message to another process and be able to receive as well.

Desired Response

- A message is sent and a message is received.

Participating Classes

- EcMpMsgCtrl, EcMpMsgCb, EcMpSessionList, EcMpMsgQueue, EcMpMsgQueueIn, EcMpMsgQueueOut, EcMpTransferSrv, EcMpTransferCli.

Pre-conditions

- The user must determine the receiver identity and under which identity the process wishes to receive messages.

Post-conditions

- The message is delivered, and the notification is passed back to the sender.
- A message is received from another process and it is retrieved.

Scenario description

a. Setup an ordinary receiver session

1. Instantiate an object of the EcMpMsgPsngCtrl type using a filename (for the outgoing queue and for the disk filename/persistence), and an application name (a full CDS path name + application name).
2. Using the message passing control object start listening for incoming messages (Initialize). Next, instantiate receiving objects, that is, create an instance of the incoming queue object (via the 'createReceiver' call) and set it to point to the queue the message passing control class is attached to it.
3. When a message comes in, it gets stored in to the queue associated with the object and a copy of it is stored on the file associated with the object.
4. In order retrieve a message two methods can be used: 1) GetMessageWait (block - there is wait) Performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call; 2) GetMessage (Non Block - there is no wait) Performs a single read operation and returns a message from the Incoming Queue or a return status (“no more messages”).

b. Setup the sending session

1. A sending session is basically a point-to-point session to send a message. Each sender session will have a logical name that is needed to contact the receiver. A list of sending sessions are maintained in a given application. A sender session list contains a callback object which provides virtual functions to be called when a send is complete. This is done at the sender side.
2. Create a callback object and implement/compile the acknowledgment virtual function. The object EcMpMsgCb has a virtual function named 'handleAck', this function is called when a message is delivered to the destination or when the underlying mechanism failed to deliver it within the given constrains(number of tries). This function takes the following arguments: a flag indicating success/failure, the number of tries, the

time between tries, the actual message, its length, a receiver address, and a destination address, and a message identifier. This same unique id is presented to the application programmer when sending a message.

3. Create a EcMpSessionList and populate it with the following information: 1) Associate the callback object just created, 2) Set the number of tries and the time between each retry, and 3) For each session give the receiver's (destination) logical name.
4. Prepare and send the message to the session list. At initialization time a number of threads were generated internally which periodically will get the messages from the outgoing queue and send them. The send call returns back a UUID so the sender that can be used by the callbacks or to get a reply back.

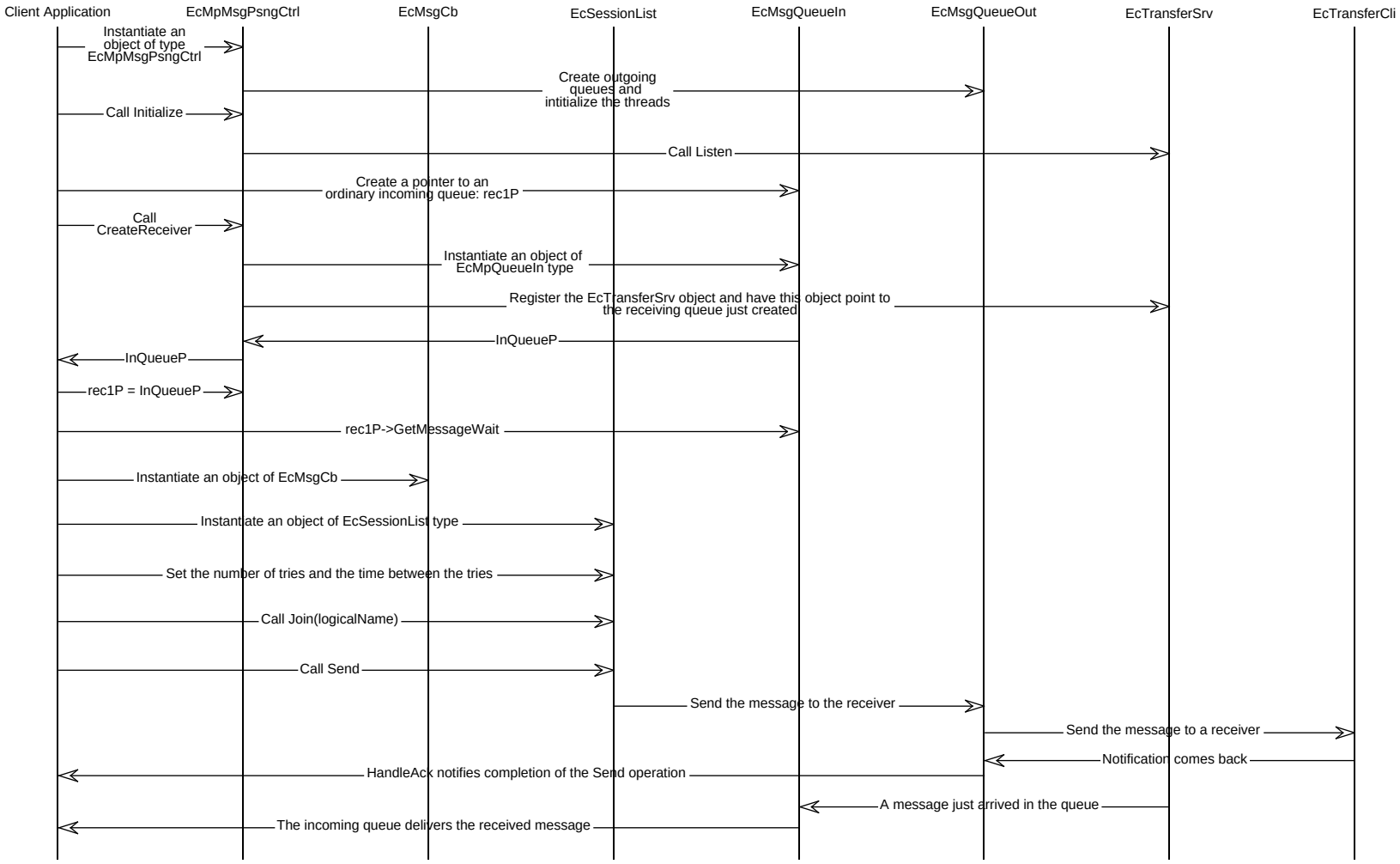

Figure 3.2.2.1.7-1. Message Passing Scenario One Event Trace

3.2.2.1.7.2 Scenario #2

Abstract

- This scenario will demonstrate the usage of a callback to receive a message. The receiver will be notified everytime a message is received.

Interfaces

- OODCE provided classes.

Stimulus

- The arrival of a message.

Desired Response

- A thread is awakened when a message is received in the queue. A callback is invoked at that point.

Participating Classes

- EcMpMsgCtrl, EcMpMsgCb, EcMpSessionList, EcMpMsgQueue, EcMpMsgQueueCbIn, EcMpMsgQueueOut, EcMpTransferSrv, EcMpTransferCli.

Pre-conditions

- The user must determine the receiver identity and under which identity the process wishes to receive messages.

Post-conditions

- A callback notification is received informing the user a message arrived.

Scenario description

a. Setup a callback receiver session
    1. Instantiate an object of the EcMpMsgPsngCtrl type using a filename (for the outgoing queue and for the disk filename/persistence), and an application name (a full CDS path name + application name).
    2. Using the message passing control object start listening for incoming messages. Next, instantiate receiving objects, that is, create an instance of the callback incoming queue object (via the createReceiverCb call) and set it to point to the queue the message passing control class is attached to it.
    3. When a message comes in, it gets stored in to the queue associated with the object and a copy of it is stored on the file associated with the object. A 'call back' thread (just one thread) will be awakened every time a message arrives into the special receiver queue created by the call 'createReceiverCb'. A call back (virtual function) will be executed. It will check if there is another message and if so execute another callback. If there are no new messages it shutdowns the thread until a new message arrives in.

Event Trace

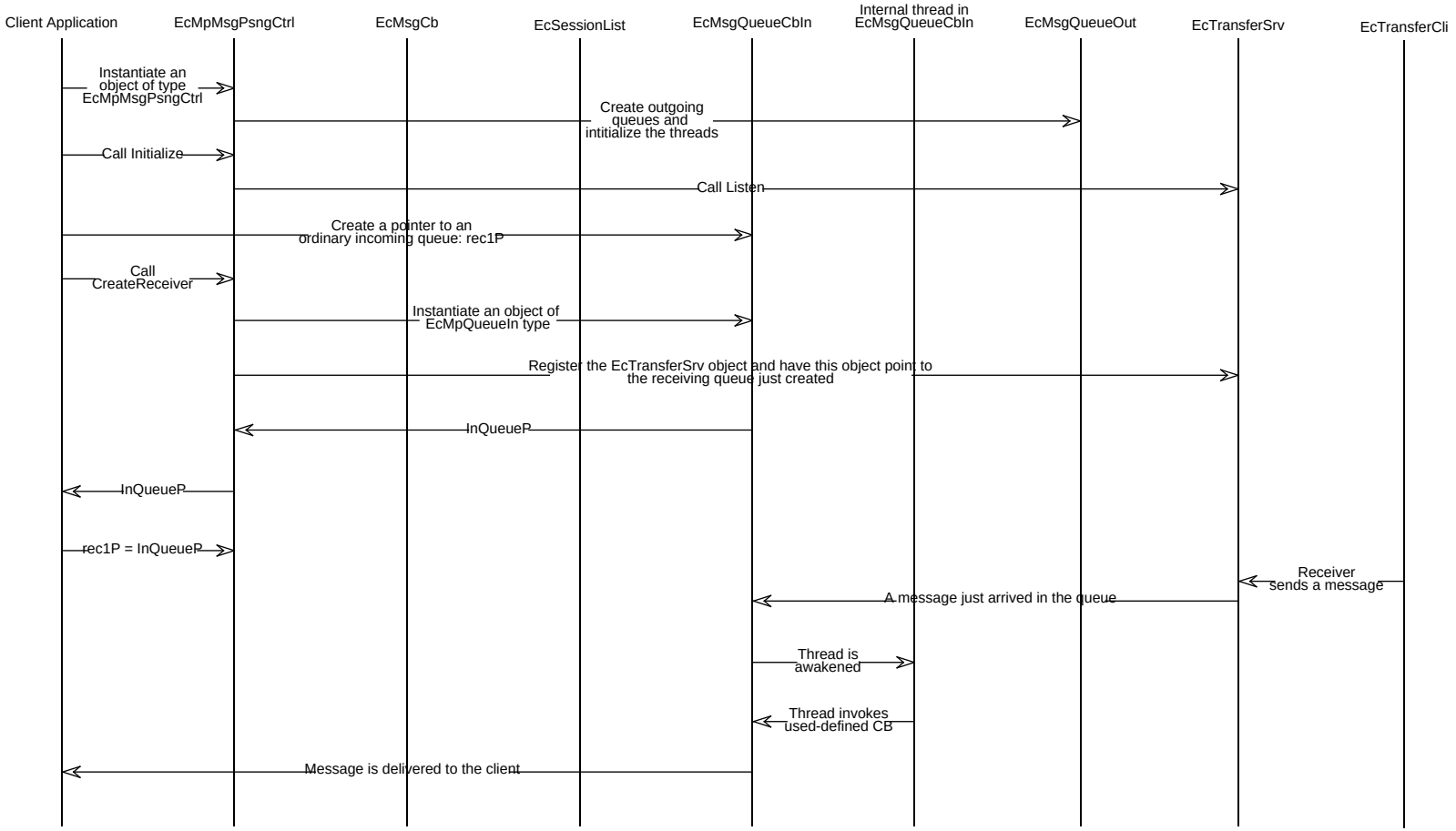

Figure 3.2.2.1.7-2 Message Passing Scenario Two Event Trace

3.2.2.2 Message Passing Implementation Two

3.2.2.2.1 Overview

Implementation two provides asynchronous and deferred synchronous communication. It is designed to work with the OODCE-provided DCE-Pthread class which is used to start and control the execution of a thread. The CSS API is not going to be as transparent to the user as the CORBA application will be, that is, some developer involvement is required. For example, with Store and Forward, the developer using the second implementation needs to create a thread to listen for a reply from the server. Reply sensing will be OTS with CORBA. In addition, recovery is not supported with the second implementation.

The CSS API spawns a thread to send a message. This thread invokes predetermined, application-programmer specializable, virtual functions. The API supports asynchronous communications.

The CSS API also supports deferred synchronous communication. The results produced by the execution of the thread can be retrieved using the GetResults member function.

The number of tries, as well as the time between each try, must be defined.

Thread scheduling attributes can be defined if the defaults are not desired. Scheduling policy controls the algorithm used to schedule threads. Scheduling priority controls the treatment of a given thread relative to other threads. The default scheduling policy is Foreground. All the threads with this policy will be scheduled on a round-robin basis regardless of their priority. Higher-priority threads will get better treatment, but all will get some time to run - to provide some fairness to low-priority threads. Foreground threads can still be locked out by higher-priority thread types, such as, FIFO or straight Round Robin. The default scheduling priority will apply to the minimum symbol of the default scheduling policy.

Only one send call can be executed at a time for each method, that is, threads will not be launched concurrently. The developer can check the status of a thread by calling 'CallInProgress'. This function will return an integer, a '0' for currently in progress, a '1' for not currently in progress.

The programmer can call the method 'Done' in order to find out whether the operation has finished successfully or not.

The 'Reset' is used to deallocate any memory assigned to the results field and to reset the flags. After Reset, the developer can call Send again or just delete the client object and terminate.

3.2.2.2.2 Context

Message Passing is an infrastructure key mechanism and is used by ECS subsystems for deferred synchronous, and asynchronous communication where a client needs control back immediately after invoking a remote procedure.

This service utilizes COTS within the OODCE and DCE products. CSS provides one generic class with virtual functions which FOS and SDPS must inherit in order to achieve asynchronous and deferred synchronous communication.

3.2.2.2.3 Directives and Guidelines

The programmer needs to define a new class inheriting from the generic EcDcDSyncCom class and should implement all the virtual functions defined in the parent class, that is, PreInvoke, Invoke and PostInvoke. In PreInvoke, the user can do any initialization that is needed prior to the transfer, Invoke executes the actual transfer method. Once control returns back from Invoke, PostInvoke is called. This method can take care of the Notifications in the case of Asynchronous message passing.

The main idea behind this API is to allow the programmer to spawn a thread for listening purposes and then release it so that the main thread is available to perform other operations such as sending data.

The programmer must do the following in order to transfer data:

- Set a data pointer to some data that will be used in the overridden Invoke member **function.** It is represented as a void pointer, to allow for various types of data.
- Set the address pointer to some address that will be used in the overridden Invoke member function. It can be a port number, an IP, a binding, a CDS name, etc. The address is a void pointer. Since the developer, will be setting this field, he/she will know how to parse it and implement it in the Invoke call .
- Call the Send method. Internally this method will create a number of threads and invoke the 'EvalThread' function which will execute the PreInvoke, Invoke and PostInvoke virtual methods. If the send fails, the call will be retried as many times as defined by the developer. Once the thread finished execution, it is terminated, but the client object will not be deleted. Results from the thread execution are returned, and 'reset' needs to be called to deallocate any memory priory set aside. The programmer can then either send some more data or terminate and delete the client object.

Assumptions:

- Suppose the client class inheriting EcDcDSyncCom had more that one method, then, for each method, an instance of the client class will be created, that is, for each one there will be one call, one thread executing at one time.

3.2.2.2.5 Sample Application Programmer Interface

Sample

This scenario demonstrates how a process can be a sender of deferred synchronous messages and at the same time operate as a server by listening for incoming requests (process sends a message to another process, and get a reply back).

Description

Step 1 First define a new class inheriting from the generic EcDcDSyncCom and implement all the virtual functions defined in the parent class, that is, PreInvoke, Invoke, and PostInvoke. Let the client class be CliDSyncCom.

For example define the class as follows,

```
Class CliDSyncCom : public EcDcDSyncCom {
```

```
        public:
            CliDSyncCom();     // Constructor
            virtual ~CliDSyncCom();// Destructor
            // Implementation of pure virtual base class functions
            virtual EcTInt PreInvoke();
            virtual EcTInt Invoke();
            virtual EcTInt PostInvoke();
    };
```

and override the virtual functions.

```
    EcTInt CliDSyncCom :: PreInvoke()
    {
            EcTInt ECSStatus;
            // ... do some something before calling Invoke
             return ECSStatus;
    }
    EcTInt CliDSyncCom :: Invoke()
    {
            EcTInt ECSStatus;
            // ...
            // Suppose there was some class called 'Telemetry' with a method called 'transfer',
            // then, we set a pointer of type 'Telemetry', 'TelP', to point to '_addr', an IP
           // address, a port number, a binding, or anything of that type.
            if ( _data)
            {
                    try     // Set up try block for exception handling
                    {
                             Telemetry *TelP = (Telemetry *)_addr;
                            // In the case of Deferred synchronous,  we want to store the results
                            // from the computation in '_results'
                             _results = TelP->transfer(....);
                    }
                    // catch any DCE related errors and print out an informative string if any
                    //occur
                      catch (DCEErr& exc)
                    {
```

```
                        // ...
                    }
                }
                return ECSstatus;
        }

        EcTInt CliDSyncCom :: PostInvoke()
        {
                EcTInt ECSStatus;
                // ... do some something after calling Invoke. This method can take care of the
                // notifications in the case of Asynchronous Message Passing
                 return ECSStatus;
        }
```

Step 2

Include the following object definitions:

```
#include "CliDSyncCom.h"   // contains the message passing object definitions
```

Step 3-12 In Main, include the client header definition file and spawn a thread for listening purposes, and start listening. Call pthread_yield to notify the Thread scheduler that the current thread will release the processor 'thread_listening'.

On the main thread create an instance of the CliDSyncCom object. Set the data pointer to some data that will be used in the overridden Invoke member function. Data can also be retrieved by means of the GetData() member function. Set the address pointer. It can be an IP, a port number, an object reference, a CDS name, etc. Optionally, set the thread scheduling attributes. If they are not set, the default values for each attribute will be used, that is, EcDcPri_min, and EcDcFg. Set the number of retries in case of exceptions/communication errors. Set the time between retries in seconds.

Call Send, a thread will get started and executed. The thread creation is done transparently, via this Send call. The developer does not have to deal with thread calls other than setting the priority or the scheduling policy if desired. Check whether the thread has finished execution successfully. Obtain the results that were a product from the thread execution.

Call Reset to set the flags to '0' and deallocate any memory used (i.e.: on results) . Delete the client object or reuse it.

```
EcTVoid main()
{
        Step 3
        // Create Clients. Instantiate a list of EcDcDSyncCom objects
```

```
EcDcDSyncComList*ClientList = new EcDcDSyncComList;
```

Step 4

```
// Create a client, an instance of the CliDSyncCom object
CliDSyncCom *client1 = new CliDSyncCom;
```

Step 5

```
// Set the data pointer to some data that will be used in the overridden Invoke member
// function.  Data can also be retrieved by means of the GetData() member function.
client1->SetData( (EcTVoid*)params);
```

Step 6

```
// Set the address pointer. It can be an IP, a port number, an object reference,
// a CDS name, etc.
client1->SetAddr( (EcTVoid*)"/.../csmscell/RelA_Apps/TelemetryTransfer")
```

Step 7 - Optional, defaults can be used.

```
// Set the thread scheduling attributes.  If they are not set, the default values for each
// attribute will be used, that is, EcDcPri_min, and EcDcFg.
EcEDcThreadPriority a_priority = EcDDcPri_low; // scheduling priority
client1->SetPriority( a_priority);
EcEDcThreadPolicy a_policy = EcDDcFifo;          // scheduling policy
client1->SetPolicy( a_policy);
```

Step 8 - Optional , defaults can be used.

```
// Set the number of retries in case of exceptions/errors
client1->SetNoOfRetries( 5);
```

Step 9 - Optional , defaults can be used.

```
// Set the time between retries in seconds
client1->SetTimeBetweenRetries( 10);
```

Step 10

```
// Send a message in Asynchronous mode. The Invoke method is executed and
// returns control back to the caller. The PostInvoke method function is
// implemented such that, once the Invoke Call completes, PostInvoke notifies
// back the caller about it. There is no wait.
// When send is called , a thread will get started and executed. The thread creation
// is transparent to the developer
client1->Send();
```

Step 11

```
// Check whether the thread has finished execution successfully.
```

```
    EcTInt job_status = Done();
    Step 12
    // Set the '_done' flag to '0' and deallocate '_results'.
    client1->Reset();
    Step 13
    // Delete the client1 object or reuse it.
} // End of Main
```

3.2.2.2.6 Object Model

Table 3.2.2.2.6-1 summarizes the second Message Passing Service classes which are discussed in detail in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, Section 4.2.3.2.

Table 3.2.2.2.6-1 Message Passing Object Responsibility Matrix

Class Name	Description
EcDcDSyncCom	This class is used to achieve message passing using asynchronous and deferred synchronous communications. It is designed to work with OODCE-provided DCE-Pthread class which is used to start and control execution of a thread. The user is expected to override the virtual member functions of this class, which are, PreInvoke, Invoke, and PostInvoke, in order to perform whatever operations are needed. Provides the functionality to – receive messages from sender – transmit them to the receivers – collect and maintain acknowledgment information from the receiver. – collect and maintain results from the receiver – pass the acknowledgment and results back to the sender.
EcDcDSyncComList	This class is used to send a message to a list of sessions in the group.

3.2.2.2.7 Dynamic Model Scenarios

3.2.2.2.7.1 Scenario #1

Abstract

- This scenario will demonstrate how a process can be a sender of deferred synchronous messages and at the same time operate as a server (listen for incoming requests).

Interfaces

- OODCE provided classes

Stimulus

- The process wants to send a message to another process, and get a reply back. It also listens for incoming requests.

Desired Response

- A message is sent and its result comes back.

Participating Classes

- EcDcDSyncCom

Pre-conditions

- The user must determine the receiver identity and under which identity the process listens.

Post-conditions

- The message is delivered, and the notification is passed back. The result comes back after the send operation completed.

Scenario description

- First define a new class inheriting from the generic EcDcDSyncCom and implement all the virtual functions defined in the parent class, that is, PreInvoke, Invoke, and PostInvoke. Let the client class be CliDSyncCom.
- In Main, include the client header definition file and spawn a thread for listening purposes, and start listening. Call pthread_yield to notify the Thread scheduler that the current thread will release the processor 'thread_listening'.
- On the main thread create an instance of the CliDSyncCom object. Set the data pointer to some data that will be used in the overridden Invoke member function. Data can also be retrieved by means of the GetData() member function. Set the address pointer. It can be an IP, a port number, an object reference, a CDS name, etc. Optionally, set the thread scheduling attributes. If they are not set, the default values for each attribute will be used, that is, EcDcPri_min, and EcDcFg. Set the number of retries in case of exceptions/ communication errors. Set the time between retries in seconds.
- Call Send, a thread will get started and executed. The thread creation is done transparently, via this Send call. The developer does not have to deal with thread calls other than setting the priority or the scheduling policy if desired. Check whether the thread has finished execution successfully. Obtain the results that were a product from the thread execution.
- Call Reset to set the flags to '0' and deallocate any memory used (i.e.: on results) . Delete the client object or reuse it.

Event Trace

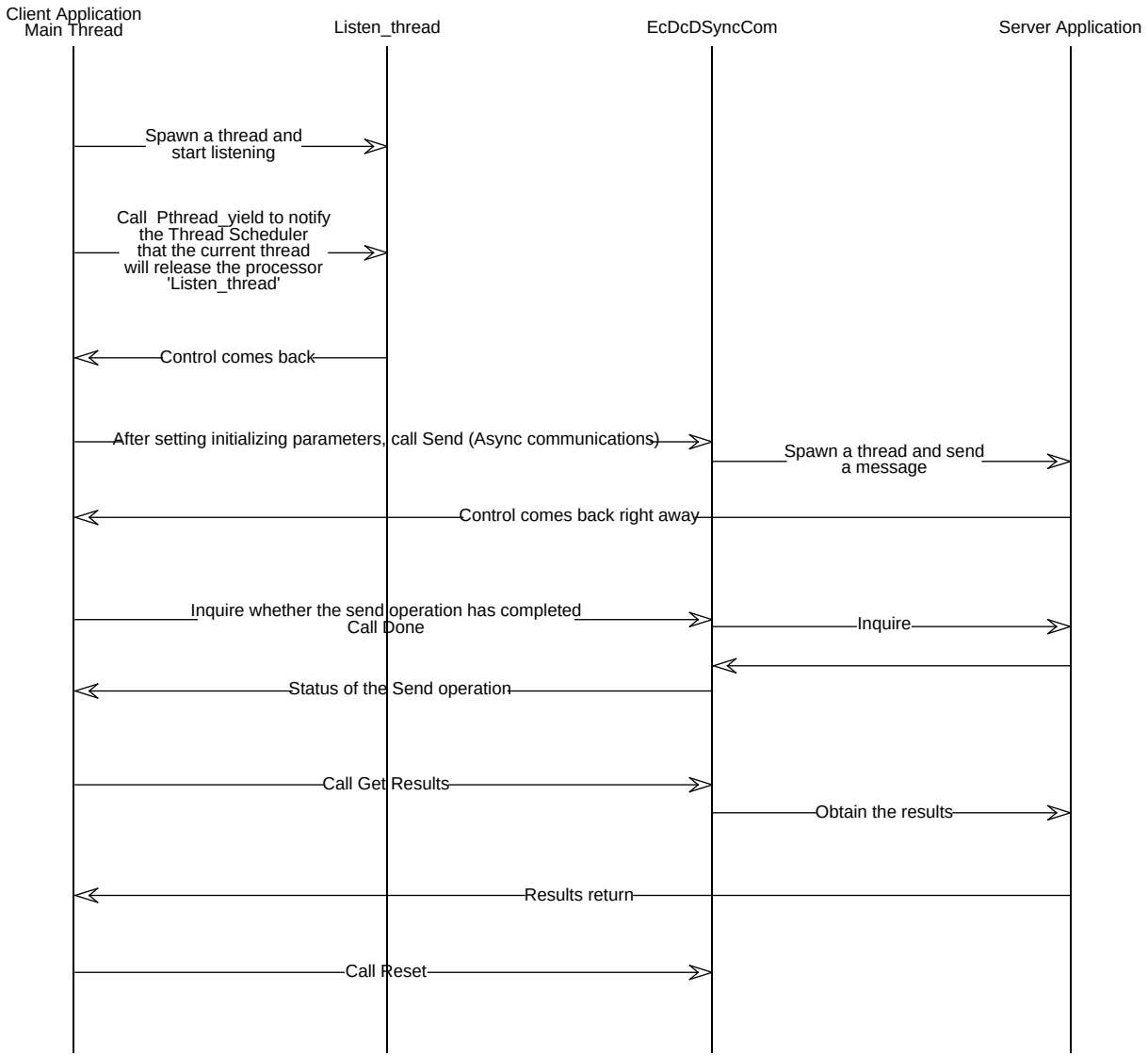

Figure 3.2.2.2.7-1. Message Passing Scenario Event Trace

3.2.3 Multicast

3.2.3.1 Overview

Multicasting is a mechanism through which a single copy of data is transferred from a single point to several places. In the point-to-point communications, data is transferred from one application to another application. If an application needs to send the same data to several other applications, the same data is needed to be sent to each of the receiving applications. Multicasting allows a sending application to specify a multicast address and send one copy of the data to that address. This data is then distributed through the Multicast backbone to all the applications listening at that address. This reduces the network traffic and improves the performance.

3.2.3.2 Multicast Context Within FOS

The multicast API(s) will be used by FOS for distributing three types of data:

1. Real-time telemetry
2. Events
3. Shared playback

FOS shall use other standard API(s) or protocols (such as TCP/UDP) for all other data types. Each of the three types is detailed below:

Real-time Telemetry

Real-time telemetry is sent from EDOS to the EOC. The data flow is unreliable and uses UDP/IP-multicast in conjunction with EDOS-provided multicast routers. The multicast IP groups will be predefined, and each group will reflect a different telemetry data stream. Thus, EOC hosts need only receive the data stream they are interested in and therefore do not require a mechanism to notify EDOS of group membership.

Each EOC Operational LAN host will receive telemetry directly from EDOS simply by listening to the appropriate pre-defined multicast address. A multicast reflector will be created for each multicast group to forward the EDOS telemetry to the ISTs.

The multicast API(s) on the EOC hosts will be used for the receipt of EDOS multicast telemetry data. The API(s) will also be used by the multicast reflector to forward telemetry to the ISTs that registered on the reflector.

Events

Events are sent from any EOC multicast capable host in response to some significant change in status (such as a host connecting to a string or generating a shared playback). The event message will be sent to every machine on the network. (For instance, if a host on the EOC Operational LAN generates an event, the event message will be sent to all EOC hosts on the Operational LAN and all ISTs with sessions to Operational LAN hosts. The event message, in this case, will not be sent to the Support LAN hosts.)

The multicast API(s) will allow for generation and receipt of events, and will handle unicasting events to the ISTs.

Shared Playback

Shared playback consists of non-real-time telemetry being sent to one or more hosts at up to 12X the real-time rate. The receiving group can consist of both ISTs and EOC hosts.

3.2.3.3 Implementation

CSMS will provide FOS with an unreliable multicast service in the form of an API(s) providing C++ classes. The multicast service shall be implemented at the transport layer via UDP and at the network layer via IP multicasts or (where necessary) unicasts. The API(s) will allow multicasting between hosts located within either the EOC Operational or Support LAN (but not between the two). Communication to the ISTs will be via unicast, but the API(s) will shield this detail from the FOS applications, so that the application makes a call to send to single group and the API insures that the data is sent to all members (whether unicast or multicast) of the group. (Note that since this is an unreliable multicast service, neither the API(s) nor the underlying protocols insure data receipt; this must be handled by the FOS application.)

3.2.3.4 Multicast API

The C++ multicast API primarily involves objects of three classes: EcMpMcCtrl, EcMpMcGrp, and EcMpMsg. The EcMpMcCtrl class is used to do the IP multicast initialization, multicast group / reflector setup, and message polling. Each EcMpMcGrp instance corresponds to a single multicast group that the application is a member of. EcMpMcGrp takes an instance of EcMpMsg class and transmits it to the group and/or constructs it in the receiving process.

The multicast APIs are defined and described below.

```
/* ---------------------------------------------------------------------
*  Multicast API Class Description
* ---------------------------------------------------------------------- */

NAME   Multicast C++ APIs

SYNOPSIS #include <EcMpMc.h>

    class EcMpMcCtrl;
    class EcMpMcGrp;
    class EcMpMsg;
```

EcMpMcCtrl Class

The EcMpMcCtrl class is used to do the IP multicast initialization, multicast group / reflector set-up, and message polling tasks.

EcTInt EcMpMcCtrl::Init();

Description: The Init() method initializes the IP multicast function, and should be called once before using other multicast APIs.

Return: SUCCESS on success; FAILURE on failure

EcTInt EcMpMcCtrl::Poll(EcTBoolean block);

Description: The poll method is used to check the receiving buffer for the incoming data. When block is set to TRUE, Poll will not return until incoming data is available, otherwise, Poll will return after the checking.

If the UNIX notification scheme is used, the Poll method should be put in the application main loop. It is more efficient in checking the receiving buffer when setting the block to TRUE. If this can not be done, the Poll should be called about every 100 milliseconds. When data is available, the corresponding HandleMsg() method will be called.

Return: SUCCESS on success; FAILURE on failure

EcMpMcGrp Class

The EcMpMcGrp class contains member variables and member functions needed to join/leave a group and send/receive messages to/from a group.

```
EcTInt EcMpMcGrp::JoinGroup(EcTChar *groupName,
                            EcTUChar TTL,     // Can be SITE or REGION
                            EcTBoolean sendonly);  // When set to TRUE, member can
                                                   // only send messages to the group,
                                                   // and will not receive messages from
                                                   // this group

EcTInt EcMpMcGrp::JoinGroup(EcTChar *groupName,
                            EcTBoolean sendonly); // Default TTL is SITE
```

Description: Before receiving or sending messages to a multicast group, the application must join a group first. Each multicast group corresponds to a set of IP Multicast Address, Port, IP TTL, and a groupName.

Two methods can be used to join to a multicast group. The first JoinGroup() method takes three arguments, the groupName, TTL (Time To Live) value, and sendonly flag. The API determines if multicast or unicast should be used to join this member by checking if "UNICAST" environment variable is set to TRUE. It then searches the namespace to find out if this groupName has been registered. If not, 'FAILURE' will be returned. Otherwise, the group setup will be made and 'SUCCESS' is returned.

The TTL value can be set to SITE, or REGION. Setting TTL to SITE will keep the multicast traffic within EOC, but the unicast traffic can still reach ISTs through the multicast reflector. If the multicast traffic needs to reach EDOS, the TTL value should be set to REGION.

When setting the sendonly flag to TRUE, the application can send data to the group, but will not receive any data from this group. If the flag is set to FALSE, the application can send and receive data from the group.

The second JoinGroup() method takes the groupName and sendonly flag. A default TTL value, SITE is used in this case.

Return: SUCCESS on success; FAILURE on failure

EcTInt EcMpMcGrp::Send(EcMpMsg *msg);

Description: Messages are sent to a group via the Send() methods with a status code returned. The argument, msg is an object of EcMpMsg. It contains an instance of the EcMpMsg to be sent. Messages are unformatted text and binary data with a maximum size of 8192 bytes. Larger messages should be broke down to smaller ones before sending.

Return: SUCCESS on success; FAILURE on failure

EcTInt EcMpMcGrp::GetMessage(EcMpMsg *msg,
EcTULongInt waittime);

Description: This GetMessage() method performs a single read operation, and wait until either a message is read from the receiving buffer, or until the waittime (in millisecond) has elapsed.

Return: SUCCESS when message is available; FAILURE on failure; TIMEOUT when the waittime has elapsed

EcTInt EcMpMcGrp::GetMessage();

Description: The GetMessage() method inform the API that the UNIX notification scheme will be used to check for the incoming messages. The application uses Poll to check the receiving buffer and will get notification of the incoming message through the callback function HandleMsg.

Return: SUCCESS on success; FAILURE on failure

EcTInt EcMpMcGrp::GetMessage(XtAppContext app_context);

Description: This GetMessage() method informs the API that the Motif notification scheme will be used to check for the incoming messages. The GetMessage() method takes Motif application context, and registers the callback function HandleMsg . The XtAppMainLoop is used to check for the incoming messages.

Return: SUCCESS on success; FAILURE on failure

virtual EcTVoid EcMpMcGrp::HandleMsg(EcMpMsg* msg);

Description: If GetMessage() method is called, the application will be notified by this virtual function when a message is received from this group.

virtual EcTVoid EcMpMcGrp::HandleReflectorFail();

Description: If unicast scheme is used to join the member to a group. When the reflector is failed, the application will get notification through this virtual function.

EcTInt EcMpMcGrp::Delete();

Description: Application can leave a group through the Delete() method.

Return: SUCCESS on success; FAILURE on failure

EcTChar* EcMpMcGrp::GetGroupName();

Description: Return the name of this group

Return: Pointer to a group name on success; NULL on failure

EcMpMsg Class

Multicast service takes an instance of EcMpMsg class and transmits it to the group and/or constructs it in the receiving process.

EcTInt EcMpMsg::SetSize(EcTUShortInt size);

Description: Application should first set the size of the message to be sent in bytes when constructing a message.

Return: SUCCESS on success; FAILURE on failure

EcTUShortInt EcMpMsg::GetSize();

Description: Get the size of the received message in bytes.

Return: Size of the received message in bytes

EcTInt EcMpMsg::SetBuffer(EcTChar *msg, EcTUShortInt size);

Description: Application fills the EcMpMsg instance with the location and size of the message to be sent. Messages are unformatted text and binary data with a maximum size of 8192 bytes.

Return: SUCCESS on success; FAILURE on failure

EcTInt EcMpMsg::SetBuffer(EcTChar *msg);

Description: Application fills the EcMpMsg instance with the location of the message to be sent. Messages are unformatted text and binary data with a maximum size of 8192 bytes.

Return: SUCCESS on success; FAILURE on failure

EcTChar* EcMpMsg::GetBuffer();

Description: Get the location of the received message.

Return: Pointer to the received message on success; NULL on failure

3.2.3.5 Multicast Object Model

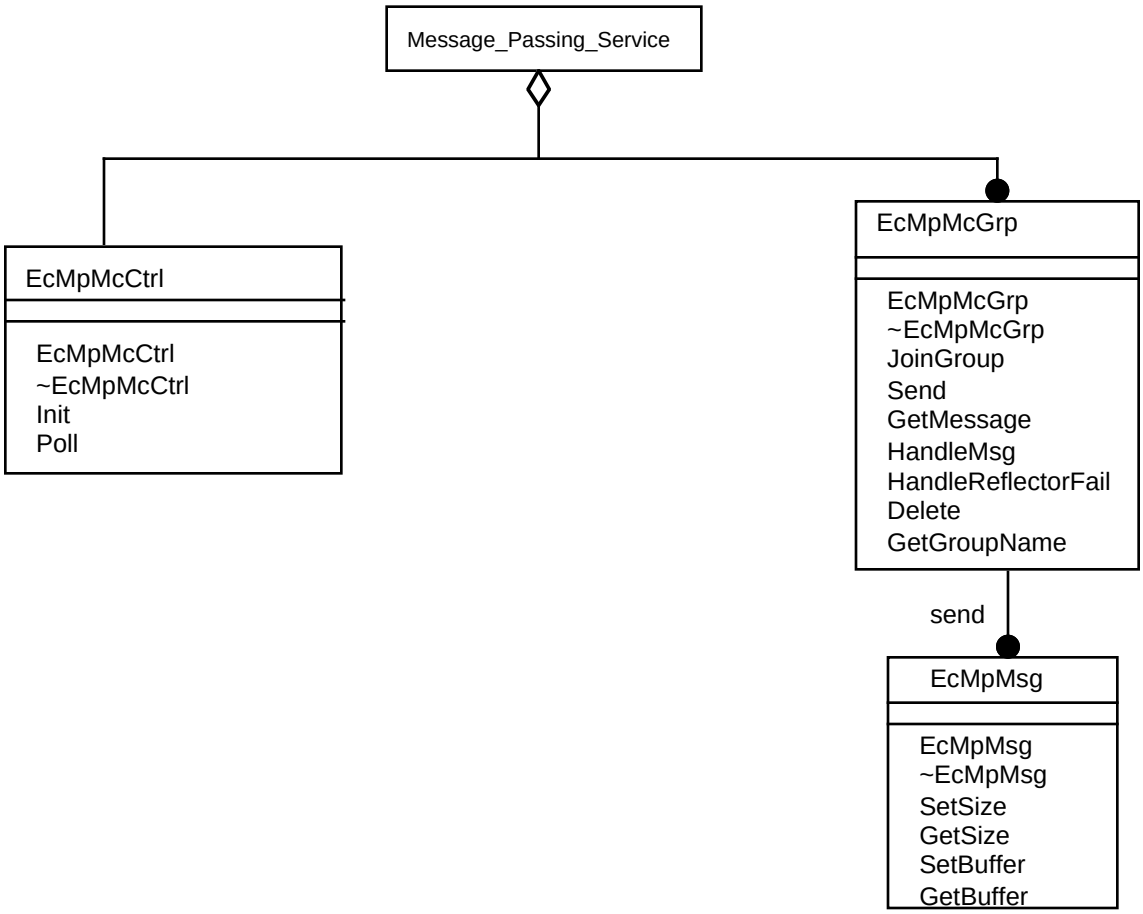

Figure 3.2.3.5-1. Multicast Object Model

3.2.3.6 Multicast Requirements

1. CSMS shall provide FOS with an unreliable multicast API(s). The data flow is unreliable and unordered at the receiving sites.
2. The multicast API(s) shall be provided as a set of C++ classes.
3. FOS requires multicasting within the EOC (on either the Support or Operational LANs, but not between the two).
4. The API(s) shall allow FOS applications to pass into it unformatted text and binary data with a maximum size of 8192 bytes.
5. The API(s) shall allow for ISTs and EOC hosts to be contained in the same group. Communication to the ISTs will be via unicast, but the API(s) shall shield this detail from the FOS applications, so that the application makes a call to send to a single group and the API sends data to all members (whether unicast or multicast) of the group.
6. The API(s) shall hide the unicast vs multicast decision and details from the FOS application.
7. The API(s) shall allow unicast and multicast communication within a single group.
8. The API(s) shall support communication (either unicast or multicast) to a group with a maximum of tbr unicast applications.
9. The API(s) shall allow a 192 kbps *[derived from 12X real-time]* data stream to be sent to a group.
10. The API(s) shall allow FOS applications to receive multicast data from EDOS.
11. The API(s) shall allow hosts to join or leave a group at any time without disrupting the group communications.
12. The API(s) shall allow multicast capable hosts to send data to a group; but the sender can choose if it wants to receive data from this group or not.
13. For asynchronous reads, the API shall provide a callback function to be called when data has arrived. For synchronous reads, the API shall provide a mechanism to specify a time to wait on read and return from the call when the time to wait has elapsed. The receiver buffer is 64K bytes. If the receiver buffer is full, the incoming data will get lost. FOS application has to control the data sending rate and check the receiving buffer fast enough to prevent this data loss.

3.2.3.7 Release Note

1. This release supports Solaris 2.x only. It is because the RoughWave Toolkit is used in the API's implementation, and we currently only have Solaris and HP RoughWave Toolkits in house; while HP-UX does not support IP Multicast without kernel modifications.
2. The multicast address information for a group is retrieved from the environment variables. Before running the multicast applications, *groupname*_ADDR and *groupname*_PORT environment variables should be defined. Next release will use DCE directory service for this multicase address lookup.

3.3 Directory Naming Service

3.3.1 Overview

The Naming Service is one of the fundamental facilities needed in distributed environments to uniquely associate a name with resources/principals along with sufficient information so they can be identified and located by the name even if the named resource changes its physical address over time. Naming is used primarily by service providers to register information about a service and by clients to locate the services.

Naming may be used more generally to store and retrieve any general information that is required to be made available about an object across a network. This information could include a server's binding information (e.g., an ECS search program that is going to search the databases for a specified criterion), file set locations (a file containing the forest vegetation for a specific time), an science product type (e.g., MODIS 2B), a network resource (e.g., a printer), information about principals (the security namespace containing user passwords, telephone numbers). The Naming service organizes this information in namespaces.

There are two widely known Name service specifications: ISO X.500 and the Internet's Directory Name Service (DNS). DCE Global Directory Service (GDS) is an implementation of the X.500 specification, and BIND is an implementation of the DNS. Of the two, BIND is more widely used. While these are standard enterprise namespaces, the interface provided for these namespaces is different and is at a very low level for the application programmer to use. X/Open had addressed the need for communication across namespaces with an interface called X/Open Federated Naming (XFN) that specifies a common interface a namespace has to support. The intent of this specification is to provide a common abstract interface that can be implemented on top of both the DNS and X.500 as well as to provide a way for the enterprise namespaces to communicate with each other.

CSS will provide an implementation of both the DNS and the X.500 namespaces. These namespaces are used to connect the local namespaces with other namespaces. CSS will provide OSF Cell Directory Service (CDS) and OSF GDS as the local namespaces. These local namespaces are used to store server binding information. Both CDS and GDS provide a standard (X/Open) application program interface called the XDS/XOM interface for the application programmers to interact with them. While namespaces are primarily used to save server information by application frameworks (like the Distributed Object Framework), they do not normally use this interface to communicate with the namespace. A specialized, more efficient internal interface is provided for these application frameworks by the local namespace (like the NS interface for CDS) to store and retrieve server binding information.

Application programmers need to use the XDS/XOM interface in order to store and retrieve application specific information into the namespaces. XDS/XOM interface is too tedious and complex to use. CSS will provide an XFN like interface to store and retrieve information in the CDS and X.500 conformant namespaces (OSF GDS is an X.500 conformant namespace). The XFN functionality will be implemented on top of the XDS/XOM interfaces. As such, the functionality provided here should work on other X.500 namespaces.

There are standard ways to startup and shutdown the DNS and the X.500 namespaces which will be provided to the M&O staff by the MSS Management applications. Starting and shutting down CDS is part of the Distributed Object Framework and is provided to the M&O staff through the MSS Management applications.

The CDS and GDS consists of entries (name and attribute value pairs). These entries may be protected through Access Control Lists (authorization). While CDS provides complete Access Control of these entries, GDS provides only a limited Access Control (authorization) which is better than UNIX OS Access Control Lists.

A name consists of a sequence of one or more contexts composed according to the naming convention, and the entry name. Each entry name is associated with a set of zero or more attributes. Each attribute in the set has a unique attribute identifier, an attribute syntax, and a set of zero or more distinct attribute values.

3.3.2 Context

Directory Naming is an infrastructure key mechanism and is used by ECS subsystems who need to use a Cell Directory namespace as a database to enter or retrieve information stored in the form of attribute-value pairs.

M&O will use an MSS application to store user profiles containing such information as a user's telephone number and office location and retrieves it. FOS Planning and Scheduling will use the namespace to save process related information such as location and messaging interest. Other applications may use the namespace to store and retrieve any data that should be location independent and be visible to several applications such as a common message queue to send messages asynchronously, a Universal Resource Locator (URL) of an object that an SDPS application uses. All of this interaction with the namespace is done via the interface provided by the CSS Naming service. Other interaction with the namespace to save and retrieve server binding information is done via the local specialized interface.

3.3.3 Directives and Guidelines

Directives:

The programmer must do the following in order to store/retrieve information from the DCE Namespace:

- Log in to the Cell Directory Namespace.
- Have read/write access to the directory path where entries will be added, modified, or deleted via the API. There is a command, the rgy_edit command, that the cell administrator must run, in order to assign rights to a particular object in the CDS.
- Create a context root name. Instantiate an EcDnContext object. The context class is used to define the type of name to be used in the DCE environment, it takes either a global root name(/...) or a cell root name (/.:) A context object is passed to the Composite Name class as an ordered sequence of components.
- Create subordinate contexts (EcDnContext type of objects).
- Instantiate an EcDnCompositeName object and add the contexts to it. The composite class allows the user to form a composite name (by means of the AddContext operation). A

composite name will represent a full path name (equivalent to a directory) or an entry in a database (equivalent to a file). The user can perform various operations on a composite name when the composite name is a directory such as list the contents, read entries, add and delete elements (an attribute/value list pair) element into this directory, and get the element types that had not been previously used.

- In order to add or delete an element to the composite name, must construct an element object using the element class. The element class provides methods to add values, get value list, delete values, modify values, and to get the element name.
- The element class makes use of the value and attribute object classes. The attribute class contains the attribute name and type, and provides methods to obtain the attribute name and type. The value class contains a value.

Guidelines:

The Directory Naming Service is used to construct large, enterprise-wide naming graphs. A name-to-object association is called a 'name binding'. It is defined relative to a 'Naming Context'. A 'Naming Context is an object that contains a set of name bindings in which each name is unique. 'Naming Contexts' represent 'directories' or 'folders' and other names identify 'document' or 'file' kind of objects.

The Context here can be thought of as a database key and the attributes are the actual information associated with the key. Each context forms a hierarchy where each context has a parent. Each namespace has a parent that may be another namespace maintained else where. There will be a root context, with no parent, through which all the contexts can be accessed.

Leaf contexts form the key to an entry in the namespace, to which information is attached. This information contains a list of attributes, each of which contain an identifier, a syntax representing the type of the attribute, and a list of values for that attribute. Each attribute type is identified by a unique number in the namespace. These unique numbers will be obtained from standard bodies so that there won't be any conflict in the unique ids used across different namespaces.

CSS Name service provides wrapper functions to map some of the XFN calls to the underlying namespaces that are supported: CDS/GDS. These wrappers are written on top of XDS/XOM interfaces. Since XDS/XOM are X/Open's standard interfaces to X.500 namespaces, the functionality provided by CSS will work on other X.500 namespaces. CSS will only support features that are supported in the underlying namespace. For example, not all namespaces support searching in the namespace, as such, a search wrapper function will not do anything when acting on such a namespace. Both the GDS and DNS are replicated and distributed namespaces.

CSS provides five custom classes that can be used by applications to utilize the Cell Directory Name Space as a database. CSS divided the naming structure into two parts: a context and a list of elements. Each element represents an attribute-value list pair. Figure 3.2.3-1 pictorially describes the CDS entry structure.

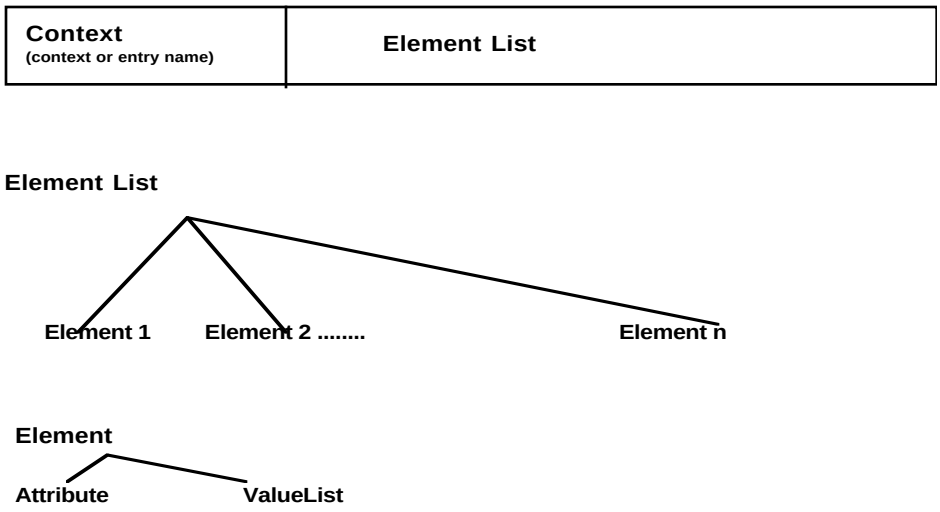

Figure 3.3.3-1. Naming Service - CDS Entry Structure

If the user wants to see graphically the contents of the Directory Naming Service, the CDS Browser is available. The CDS Browser (a GUI interface) allows a user to view the contents and structure of the cell namespace. The Browser can both display an overall directory structure, and show the contents of directories. It can also be customized to display only a specific class of object names.

Since the CDS Browser is not available on all the platforms, the 'cdscp' command can be used to view the contents of the cell namespace. 'cdscp' allows one to view selected contents of one directory at a time. It is an interactive command, not a GUI interface. For example, to display all objects in the directory ' /.:/common/dev/store ', the user can enter the following:

cdscp=>l obj /.:/common/dev/store/*

For more information regarding CDS Browser or the 'cdscp' command, please refer to the DCE administration guide.

Regarding the practical limit on the size of a DCE cell, it is still a bit early in the product life to have substantial experience with large-scale DCE installations. But there are some large cells in operation. Certainly it is reasonable to plan on cells with at least thousands of nodes and perhaps tens of thousands of users.

The University of Michigan Center for Information Technology Integration has done a study in which they added 50,000 entries to the Cell Directory and to the security registry. Their results are reported in Technical Reports 93-12 and 94-1.

Regarding how much memory and disk space is required for DCE services, this depends on the size of the cell, the number of users, number of services, etc. According to a paper present by Dan Hamel of Transarc, at the Decorum conference in February 1994, the following can be used as rough guidelines:

- Security server: 2k per principal/account; same at replicated sites

- Directory server: 10k per directory, 1k per object; same at replicated sites
- End-user machines: Each dce_login creates new credential files, which can build up. Space usage can range from less than 1k to over 100k.

Regarding whether a machine can be a member of more than one DCE cell, this is not possible at present. A machine can only be in a single cell under DCE 1.0.3. However, it will be possible for cells to cooperate when DCE 1.1 is deployed. See the next question.

Present Available

- Implementations of both DNS and X.500 are available. DNS/BIND is available in public domain from the Internet at no cost. CSS will provide this public domain BIND namespace. Implementations of X.500 are available as COTS products. GDS is one such implementation that comes as bundled software with DCE. Since DCE (OODCE) is the chosen infrastructure, CSS will provide GDS as the X.500 conformant namespace. Both BIND and GDS are replicated and distributed namespaces that support local client caching. They both provide some degree of security.

Future Available

- X/Open is currently developing code to support the XFN interface. This standard will be implemented in Release B, provided that it is ready in that time frame. In Release A, CSS will develop the required XFN functionality (with the exception of search) on top of the X/Open XDS/XOM interface for the X.500 namespaces.

3.3.4 Sample Application Programmer Interface

Following are the methods the application's developer uses to store/retrieve/list elements in a namespace.

Sample #1

List the contents of a Directory Service

Description

In order to create a composite name the user enters a series of atomic names that are in accordance with the CDS naming conventions. First create a context root name. The developer must define the type of name to be used in the environment, either a global root name or cell root name. Second, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name.

Then, proceed to list the entries/directories in that composite name; using the composite name object call the method 'listCtx'.

```
// Create a context root name
EcDnContext *ctxP = new EcDnContext(1);// where 1 is for /.:
// Create subordinate contexts
EcDnContext *ctx_1P = new EcDnContext("hp", 0);    // 0 indicates "hp" is a directory
                                                   // not a file
```

```
EcDnContext *ctx_2P = new EcDnContext("examples", 0);// 0 indicates          //
"examples" is a
                                                        // directory not a file
// Instantiate an object of type EcDnCompositeName and add the contexts to it
EcDnCompositeName *comp_nameP = new EcDnCompositeName(ctxP);
status = comp_nameP->add_ctx(ctx_1P);// we now have /.:hp
status = comp_nameP->add_ctx(ctx_2P);// and now we have /.:/hp/examples
// List the Directory Service
status = comp_nameP->list_ctx(&lstP);// Obtain contents of  /.:/hp/examples
                                        // output from list_ctx is lstP which
                                        // is  a pointer to a linked list of
                                        // containing the immediate          //
subordinates of /.:/hp/examples
```

Sample #2

Add an element into the Directory Service. The user defines the entry name, type and its respective attribute name, type and value. Makes a request to add an element.

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Proceed to add the element. Instantiate an attribute, instantiate an element, create and instantiate values, add values to the element, and add element into the Directory Service. Using the composite name object call the method 'addElement'.

```
// Create context root name
EcDnContext *ctx_AP = new EcDnContext(1);// where 1 is for /.:
// Create subordinate contexts
EcDnContext *ctx_1AP = new EcDnContext("hp", 0);// 0 indicates "hp" is a
                                                        // directory not a file
EcDnContext *ctx_2AP = new EcDnContext("examples", 0);// 0 indicates          //
"examples" is a
                                                        // directory not a file
EcDnContext *ctx_3AP = new EcDnContext("sleeper", 1);// 1 indicates "sleeper" //  is  a
file not a                                                      // directory
// Instantiate an EcDnComposite name and add the contexts to it
EcDnCompositeName *comp_nameP = new EcDnCompositeName(ctx_AP);
status = comp_nameP->add_ctx(ctx_1AP);// we now have /.:/hp
```

```
        status = comp_nameP->add_ctx(ctx_2AP);// add on and get                    //      /
.:/hp/examples
        status = comp_nameP->add_ctx(ctx_3AP);          // final name is
                                                        //  /.:/hp/examples/sleeper
        // Instantiate an attribute
        EcDnAttribute *attr1 = new EcDnAttribute("CSSAttr");// new attribute
        // Instantiate an element
        EcDnElement *elt1 = new EcDnElement(attr1);// This element is for the
                                                        // new attribute "CSSAttr"
        // Create/instantiate values
        EcDnValue *val1 = new EcDnValue("CSSValue1");// 1st value for "CSSAttr"
        EcDnValue *val2 = new EcDnValue("CSSValue2");// 2nd value for "CSSAttr"
        // Add values to the element
        status = elt1->add_value(val1);// Add the values into the element
        status = elt1->add_value(val2);// for the new attribute "CSSAttr"
        // Add element in the Directory Service
         status = comp_nameP->add_element(elt1);// now add the element to
                                                        // /.:/hp/examples/sleeper
```

Sample #3

Read elements from the Directory Service.

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Proceed to read an element. Using the composite name object call the method 'getValueList'. This operation will return the element information (the list of attributes and its values).

Delete the value object from the element and the element from the composite name. Deallocate any memory used.

```
        status = comp_name->read_element(&elt_listP);// Composite name contains
                                                        //  /.:/hp/examples/sleeper
                                                        // Output from read_element is a
                                                        // pointer to a linked list of
                                                        // element objects
                                                        // (attributes and their values)
```

Sample #4

This scenario describes how to obtain a list of values pertaining to an element (attribute).

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Given an element, specify the particular attribute to be read. Using the element name object call the method 'getValueList'. This operation will return the a list of the attribute values.

Delete the value object from the element and the element from the composite name. Deallocate any memory used.

```
status = elt1->get_value_list(&val_listP); // elt1 points to a specific
                                           //element for which we want
                                           // to obtain all the values
                                           // Output will be val_listP
                                           // which is a pointer to a        //
linked list of value objects
```

Sample #5

Delete a value from the element

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Given an element delete its value.

```
status = elt1->delete_value(val_obj);  // Delete the value object
                                       // from the element
```

Sample #6

Delete element from the Directory Service

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Delete the given element.

```
status = comp_name->delete_element(elt1);// Delete element elt1 from
                                         // the composite name -
                                         //  /.:/hp/examples/sleeper
```

3.3.5 Object Model

Table 3.3.5-1 summarizes the Directory Naming Service classes which are discussed in detail in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, Section 3.3.

Table 3.3.5-1. Naming Service Object Responsibility Matrix

Class name	Description
EcDnContext	The EcDnContext class defines the path/set of bindings with distinct atomic names. Every context has an associated naming convention. A context object is passed to the EcDnCompositeName object in a structural form as an ordered sequence of components. It is the key part of the entry in a namespace that is used to uniquely identify it.
EcDnCompositeName	The EcDnCompositeName class defines a composite name, which is a nested set of contexts in a given hierarchy concatenated together to establish a Directory Service path name. This class provides functionality to create/modify/maintain context part of an entry in the namespace. It provides methods to concatenate contexts, list the contents of the composite name in the Directory Service (soft links/object entries), read entry names, add elements (attribute/value list pair), read element information, and delete elements.
EcDnElement	The EcDnElement class contains an element, which is an attribute-value list pair. It is referenced by the EcDnCompositeName class. This class provides methods to add value(s), get value list, delete value(s), modify value(s), and get the element name.
EcDnAttribute	The EcDnAttribute class contains an attribute name, and type. It is also referenced by the EcDnElement class. This class provides methods to get the attribute name and type.
EcDnValue	The EcDnValue class contains a value. It is also referenced by the EcDnElement and EcDnAttribute class.

3.3.5.1 Dynamic Model Scenarios

3.3.5.1.1 Scenario #1

Abstract

- This scenario describes how to list the contents of the Directory Service.

Interfaces

- Cell Directory Service

Stimulus

- The user enters a series of atomic names in order to create a composite name that is in accordance with the CDS naming conventions, and makes a request to list the contents of that directory.

Desired Response

- The user will receive the contents of the composite name.

Participating Classes

- EcDnContext, EcDnCompositeName.

Pre-conditions

- The composite name must be established.

Post-conditions

- This doesn't change the state of the naming database. It only retrieves information about the composite name.

Scenario description

- Create a context root name. Define the type of name to be used in the environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name. Proceed to list the entries/directories in that composite name, that is, using the composite name object call the method 'listCtx'.

Event Trace

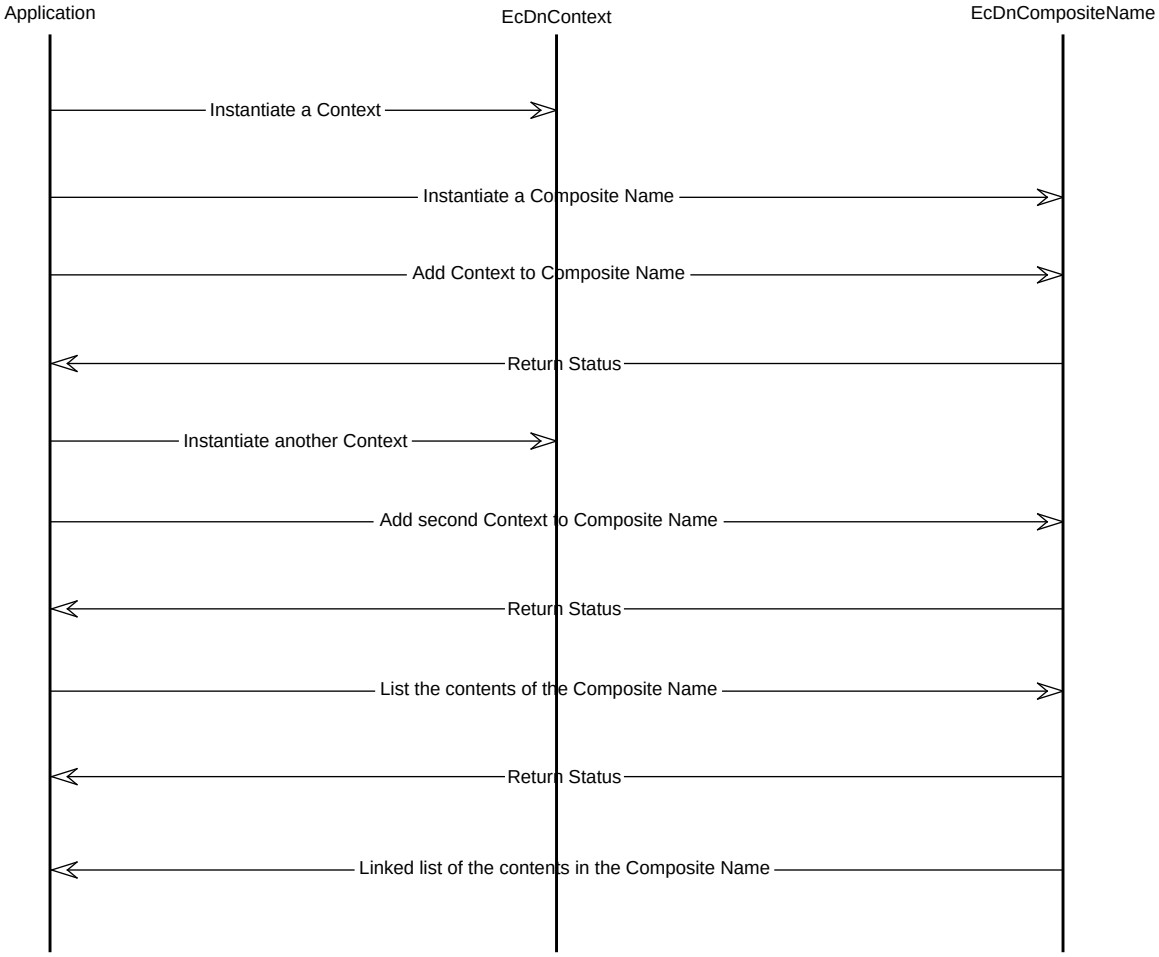

Figure 3.3.5.1-1. Naming Scenario #1

3.3.5.1.2 Scenario #2

Abstract

- This scenario describes how to add an element into the Directory Service.

Interfaces

- Cell Directory Service

Stimulus

- The user defines the entry name, type and its respective attribute name, type and value. Makes a request to add an element.

Desired Response

- The user will receive an status back after calling the add operation. It will indicate success or failure of the operation.

Participating Classes

- EcDnContext, EcDnCompositeName, EcDnElement, EcDnAttribute, EcDnValue.

Pre-conditions

- The entry name, its type, the attribute name, its type and value must be established.

Post-conditions

- This change the state of the naming database. It adds information into the CDS.

Scenario description

- Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Proceed to add the element. Instantiate an attribute, instantiate an element, create and instantiate values, add values to the element, and add element into the Directory Service. Using the composite name object call the method 'addElement'.

Event Trace

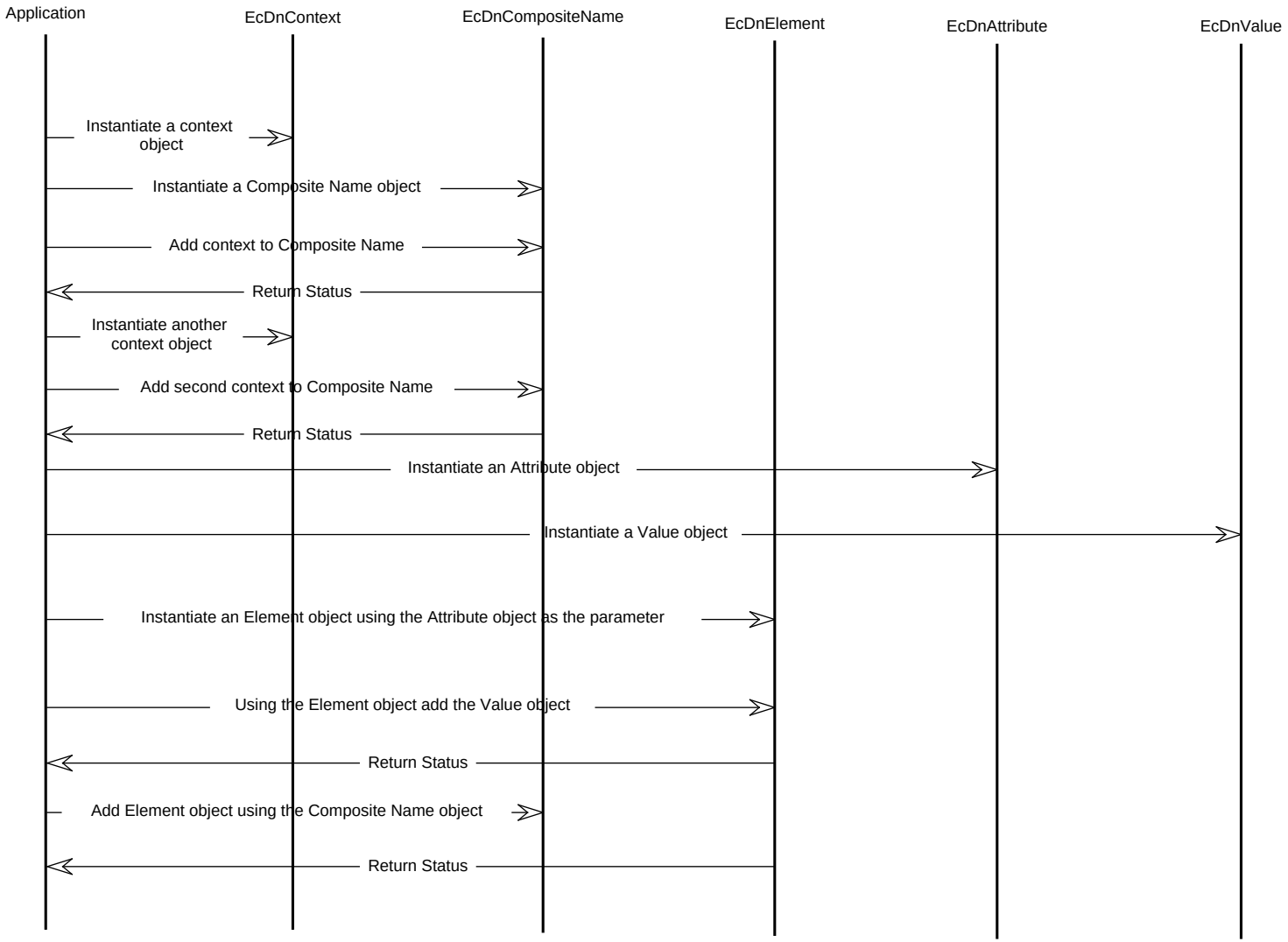

Figure 3.3.5.1-2. Naming Scenario #2

3.3.5.1.3 Scenario #3

Abstract

- This scenario describes how to read an element from the Directory Service.

Interfaces

- Cell Directory Service

Stimulus

- The user enters a series of atomic names in order to create a composite name that is in accordance with the CDS naming conventions, and makes a request to read the contents of an entry (i.e.: a file).

Desired Response

- The user will get the contents of the entry (element) read, that is, the attribute list information pertaining to that element.

Participating Classes

- EcDnContext, EcDnCompositeName, EcDnElement, EcDnAttribute, EcDnValue.

Pre-conditions

- The composite name and entry to be read must be determined.

Post-conditions

- This doesn't change the state of the naming database. It only retrieves information about the entry name. It returns a linked list of element objects (attribute/value list information).

Scenario description

- Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Proceed to read an element. Using the composite name object call the method 'getValueList'. This operation will return the element information (the list of attributes and its values).
- Delete the value object from the element and the element from the composite name. Deallocate any memory used.

Event Trace

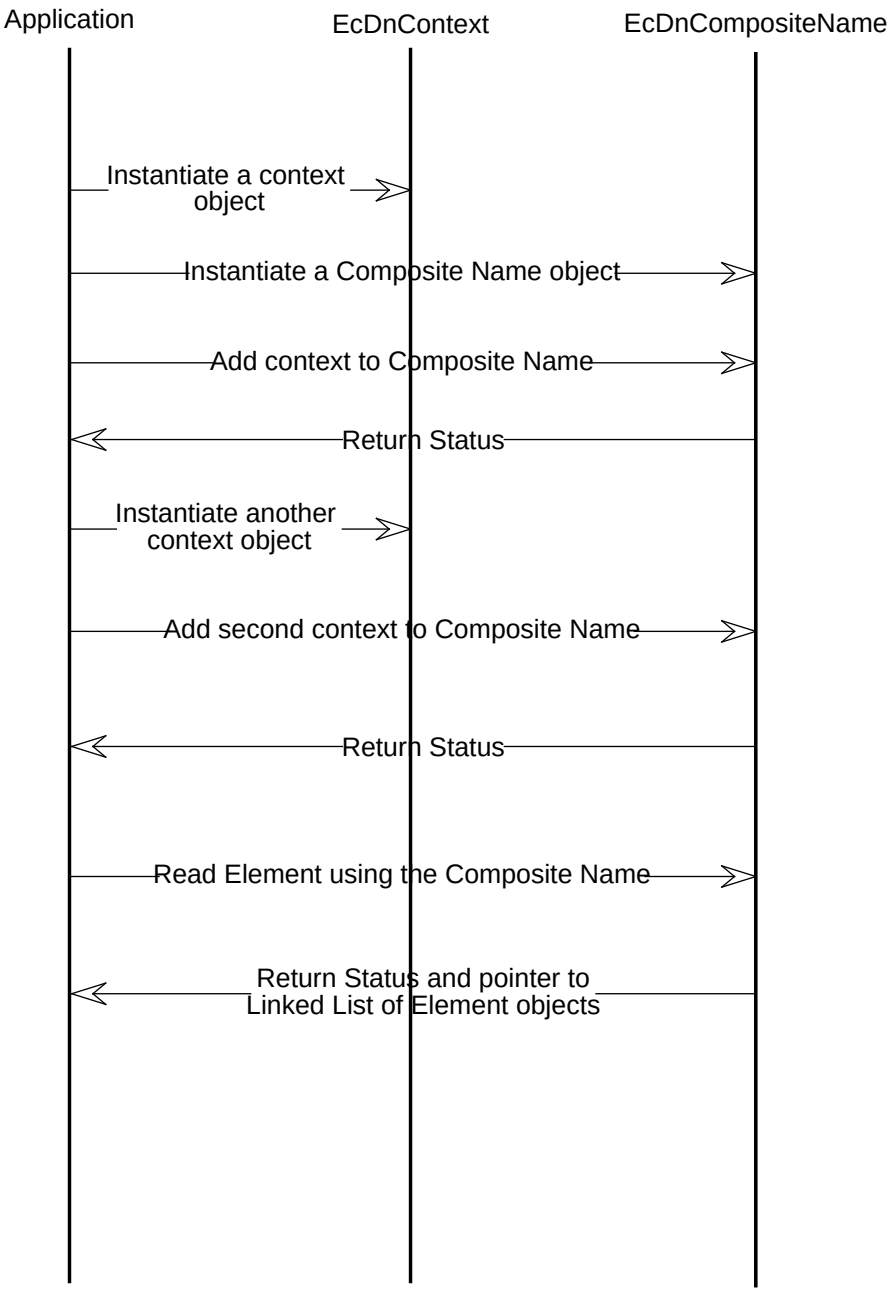

Figure 3.3.5.1-3. Naming Scenario #3

3.3.5.1.4 Scenario #4

Abstract

- This scenario describes how to obtain a list of values pertaining to an element (attribute).

Interfaces

- Cell Directory Service.

Stimulus

- The user enters a series of atomic names in order to create a composite name that is in accordance with the CDS naming conventions, and makes a request to list the values of an element (attribute).

Desired Response

- The user will list the contents of the composite name.

Participating Classes

- EcDnContext, EcDnCompositeName, EcDnElement, EcDnAttribute, EcDnValue.

Pre-conditions

- The composite name and entry to be read must be determined.

Post-conditions

- This doesn't change the state of the naming database. It only retrieves information about the element. It returns a linked list of value objects pertaining to the attribute/element.

Scenario description

- Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Given an element, specify the particular attribute to be read. Using the element name object call the method 'getValueList'. This operation will return the a list of the attribute values.
- Delete the value object from the element and the element from the composite name. Deallocate any memory used.

Event Trace

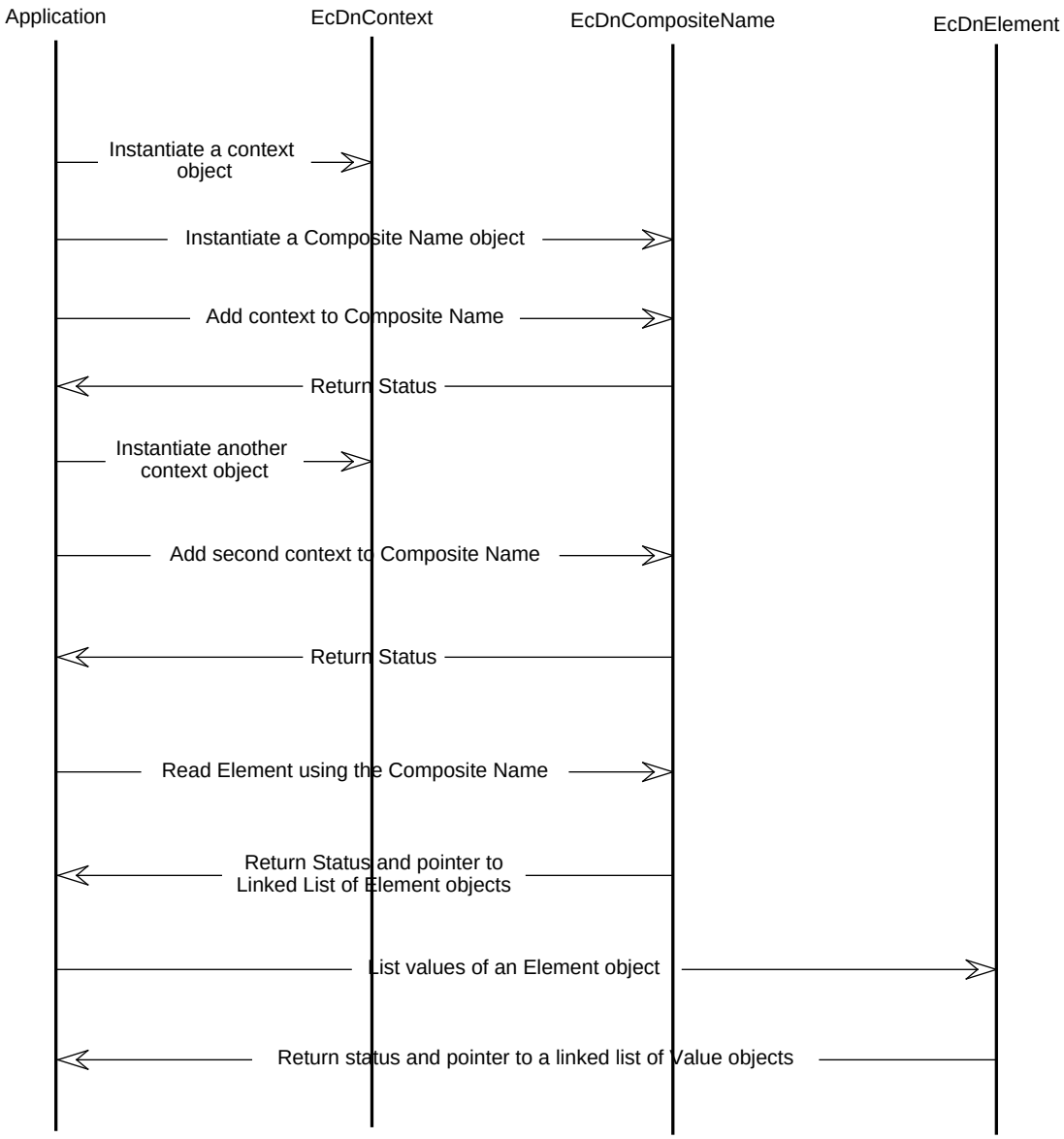

Figure 3.3.5.1-4. Naming Scenario #4

3.4 Time Service

3.4.1 Overview

The CSS Time Service will utilize the DCE (Distributed Computing Environment) DTS (Distributed Time Service) to keep system clocks in the ECS network approximately in sync by adjusting the time kept by the operating system at every node. Timestamps are used by many applications when recording event occurrences to a log. The implementation detail of the CSS Time Service and DCE DTS are invisible to the software developer.

The CSS Time Service will take advantage of the DCE DTS which has a Time Provider Interface (TPI). The TPI will allow an external time source to connect to the Time Service. A Time Provider provides access to standardized or government controlled time devices such as radios, satellites, or telephone lines. The servers with a Time Service query the Time Providers for the current time and can pass the standard Coordinated Universal Time (UTC) time values to a DTS server and propagate them through the network. The Time Providers are considered the most accurate source of time information.

The Distributed Time Service (DTS) synchronizes the system clock on each host by directly adjusting the time kept by the operating system. Under ordinary circumstances, this is done gradually so that there are no sudden jumps in the time. It is also done in such a way that the time never goes backward. If a system clock is too far ahead, it is slowed down until the time is correct by modifying the tick increment.

3.4.2 Context

All segments are expected to use the Time Service for Release A. The CSS Time Service will provide distributed time with millisecond resolution. Applications utilize the Time Service when they need to obtain the time in various formats. The Time Service provides APIs to perform these categories of functionality.

3.4.3 Directives and Guidelines

Since the CSS Time Service utilizes the DCE DTS APIs the developer's process must be run in a DCE cell which contains DTS.

The developer must instantiate an EcTiTimeService object to use the CSS Time Service.

The CSS time service will not provide a method to set time but will provide methods to obtain the time in various formats.

A delta value must be placed in the namespace before instaniating an EcTiTimeService object if a delta is to be applied to the current time. Some applications may need to simulate the current time by applying a delta to the current time. The time class allows application developers to obtain the current time in various formats and optionally lets them apply a predetermined delta to those values. The delta value in the namespace should be a byte stream having the following format

[+|-]dd:hh:mm:ss where

[+|-] indicates plus or minus
dd indicates number of days
hh indicates hours
mm indicates minutes
ss indicates seconds

3.4.4 Sample Application Programmer Interface

A few samples of how to use some of the available EcTiTimeService methods are listed below. More methods for the EcTiTimeService class are described in further detail in the CDR documentation (Release A CSMS Communications Subsystem Design Specification for the ECS Project, Section 4.2.6).

Sample #1

Instantiate an EcTiTimeService object using a delta value

Description

If the developer wishes to use a simulated time, the developer needs to first create an entry in the namespace and set the value of the delta to be used to create a simulated time (refer to CSS Directory Services to create an entry in the namespace)

The developer may then instaniate the EcTiTimeService object by doing the following:

```
EcTiTimeService*  ECSTimeP = new EcTiTimeService("NamespaceString");
```

Note: "NamespaceString" is the string for the name of the entry in the namespace where a delta value can be obtained.

Sample #2

Instantiate an EcTiTimeService object without using a delta value

Description

The developer may then instaniate the EcTiTimeService object by doing the following:

```
EcTiTimeService*  EcTiTimeServiceP = new EcTiTimeService("\0");
```

Sample #3

Obtain current ASCII GMT time

Description

First the developer must instantiate an EcTiTimeService object.

```
EcTiTimeService*  EcTiTimeServiceP = new EcTiTimeService("\0");
```

The developer should then call the GetAscGmtTime method as follows:

```
EcTChar*TimeString;   // character string to receive the time
ECSStatus = EcTiTimeServiceP --> GetAscGmtTime(TimeString);
```

Return value TimeString contains "(1995-05-16-13:23:31.215+00:00I-----)"

Sample #4

Obtain seconds and nanoseconds time values

Description

First the developer must instantiate an EcTiTimeService object.

```
EcTiTimeService*  EcTiTimeServiceP = new EcTiTimeService("\0");
```

The developer should then call the GetSecNanoTime method as follows where seconds has been declared as unsigned long and nanoseconds has been declared as long:

```
ECStatus = EcTiTimeServiceP --> GetSecNanoTime(seconds,  nanoseconds);
```

3.4.5 Object Model

Table 3.4.5-1 summarizes the time class which is discussed in detail in the CDR documentation (Release A CSMS Communications Subsystem Design Specification for the ECS Project, Section 3.4.5).

Table 3.4.5-1. Time Service Object Responsibility Matrix

Object	Responsibility
EcTiTimeService	Retrieving timestamp information Converting between binary timestamps and ASCII representations

3.4.6 Dynamic Model Scenario

Abstract

- An application wishes to obtain a simulated time.

Interfaces

- Time Service

Stimulus

- An application constructs an EcTiTimeService object with a delta value to be used for simulated time and then makes a request for the time.

Desired Response

- Application receives the correct simulated time.

Participating Classes

EcTiTimeService

Pre-conditions

- The application must place an entry in the Directory Service which contains a delta value to be used in simulating time. The application must also construct an EcTiTimeService object using the "Name" in the Directory Service where the delta value can be found.

Post-conditions

- The application receives the correct simulated time and continues processing.

Scenario Description

- The application places an entry in the Directory Services that contains the delta value and constructs an EcTiTimeService object using the "Namespace" where the delta value was placed. The application than makes a request to obtain the ASCII GMT time.

Event Trace

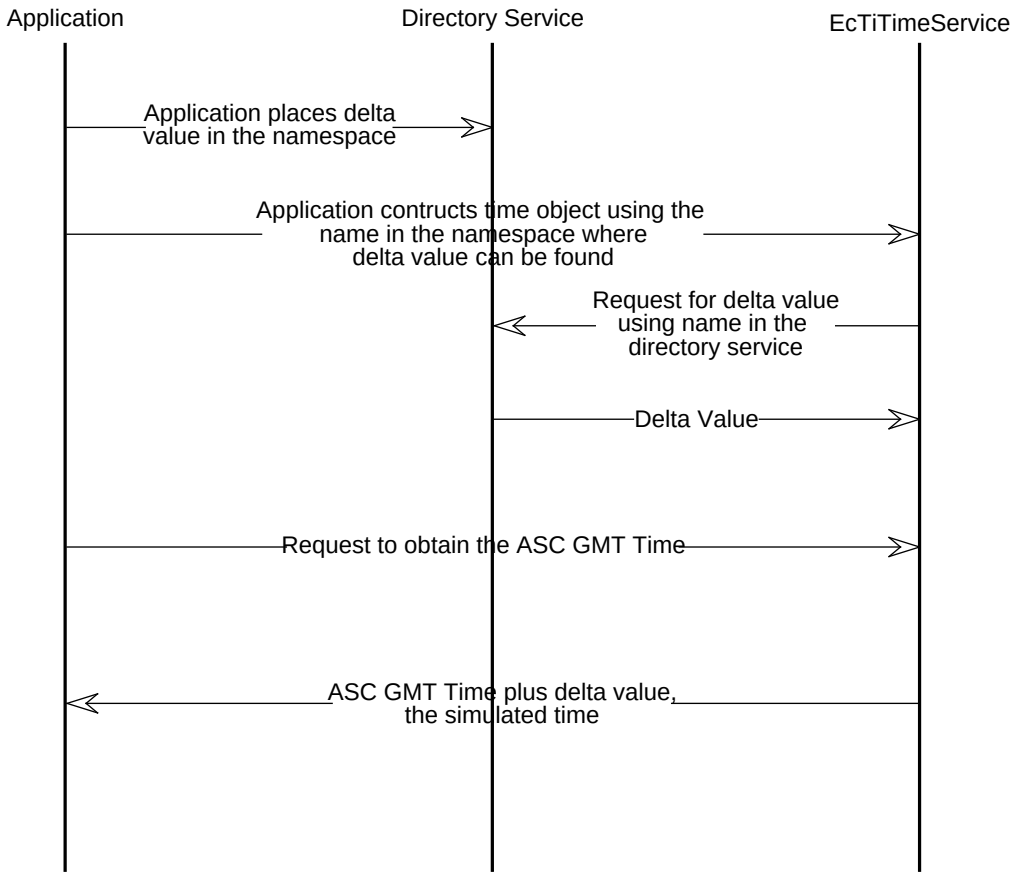

Figure 3.4.6-1. Time Service Event Trace

3.5 User Authentication

This section contains the description of the User Authentication Service and the User Account Management Service, which is provided by the Science and Communications Development Office (SCDO) and is used by FOS.

3.5.1 User Authentication Description

SCDO will provide FOS with an authentication service that will verify the identity of users when logging in. The service will return a status that will be used to accept or reject the user's attempt to login to the system.

SCDO will provide FOS with a user account management service that will allow authorized users to manage the user information, including user name, password, and user role.

3.5.2 User Authentication Context Within FOS

The authentication service will be used by FOS to verify the user name and user password when an attempt is made to login to the system. The user's login request will be allowed or rejected depending on the returning status of the authentication check.

The authentication service also shall keep information (user name, role) about all users currently logged onto the system. This service will be used by the FOS email utility to send messages to the users currently logged onto the system based upon the user's role.The user account management service will be used by FOS authorized persons to add, delete, and modify the user's account. An authorized user will use this service to add a user to the system, to delete a user from the system, or to modify the user's role information.

The available user roles (types) are :

command activity controller,

command management analyst,

database manager,

flight systems engineer,

ground controller,

ground systems engineer,

instrument controller,

instrument engineer,

instrument evaluator,

instrument planner,

mission planner,

mission supervisor,

operations coordinator,

ops controller,

shift supervisor,

spacecraft activity controller,

spacecraft evaluator,

spacecraft engineer,

spacecraft planner,

system specialist,

software maintenance engineer,

IST user,

EOC user.

3.5.3 User Authentication API

3.5.3.1 Introduction

This section describes the Authentication API to verify the user name and password. This API uses DCE security to authenticate a user. Additional APIs will be required to meet the remaining user authentication requirements identified in Section 3.4.5 of this ICD.

The Authentication API uses DCE security to authenticate a user:

```
EcTVoid
CsDcChkAuthn(
        EcTChar *principalName,
        EcTChar *principalPasswd,
        EcTInt  *loginStatus);
```

INPUT:
principalNamePrincipal Name on the registry account.
principalPasswdPassword to be checked against the password
in the principal's registry account.
OUTPUT:
loginStatusA pointer to the completion status. The following
status codes are returned:

1. The sec_login_setup_identity( ) sets up the user's network identity,
2. the sec_login_validate_identity( ) validates the login context established by sec_login_setup_identity( ),
3. the sec_login_certify_identity( ) certifies the security server used to setup and validate a login context is legitimate,
4. and finally, the sec_login_set_context( ) sets the network credentials specified by the login context.

The following error staus codes are returned in the "loginStatus" variable.

CsCDcAuthSuccess	This call was successful.
CsCDcSecLoginSetupIdFailed	The identity has not been successfully established.
CsCDcSecLoginValidateIdFailed	The identity has not been successfully validated.
CsCDcSecLoginAcctExpired	The account has expired.
CsCDcSecLoginCertifyIdFailed	Certification was not successful.
CsCDcSecLoginSetContextFailed	Unable to create network credentials for a login
CsCDcMemoryAllocFailed	Memory allocation Failed.

3.5.3.2 Description

The CsDcChkAuthn( ) API authenticates a user as follows:

1. The sec_login_setup_identity( ) is invoked, which takes the user's principal name as one of its arguments. This call causes the client Security runtime to request a TGT and passes the user's name to the Authentication service. A TGT enables a principal to be granted a ticket to the Privilege Service.
2. Upon receiving the request for a TGT, the Authentication Service obtains the user's secret key from the registry database. Using its own secret key, the Authentication Service encrypts the user's identity, along with a conversation key, in a TGT. The Authentication Service seals the TGT in an envelope that is encrypted using the user's secret key. The envelope also contains the same conversation key that is encrypted in the TGT, and is returned to the client.
3. When the TGT envelope arrives, the CsDcValidateCertifyIdentity() is invoked, which takes the user's password. This call passes the password to the local Security runtime library. The runtime derives the user's secret key from the password, and uses it to decrypt the envelope. The envelope reveals the conversation key, but the security runtime cannot decrypt the TGT, since it does not know the authentication service's secret key.
4. When the Security client runtime has succeeded in decrypting the envelope, the API calls a network layer interface that requests a Privilege-TGT (PTGT) from the Privilege service. For a PTGT to be granted, however, the user must first acquire a ticket to talk to the Privilege service, which is a principal distinct from the Authentication service. The security runtime therefore requests such a ticket from the Authentication service. The Security runtime encrypts this request using the conversation key it learned when it decrypted the TGT envelope.
5. Since the request for a ticket to the Privilege Service is encrypted under th conversation key associated with the TGT, the Authentication Service believes that the identity of the user has been established. Since the user has proved to the Authentication Service knowledge of the key, the Authentication Service allows the user to talk to the Privilege service, and so prepares a ticket to that service. This ticket contains the identity of the user (and a second conversation key) encrypted under the secret key of the Privilege service. Like the TGT envelope, the envelope containing the ticket to the Privilege service also contains the second conversation key, for use in conversing with the Privilege service, and is encrypted with the first conversation key.
6. Upon receipt of the envelope containing the ticket to the Privilege service, the Security runtime decrypts the envelope using the first conversation key, and in the process learns the second conversation key. The client RPC runtime send the Privilege Service ticket to the Privilege service.
7. The Privilege service decrypts the icket sent to it learning both the identity of the user and the conversation key it will use to encrypt its response. Because the Privilege service trusts the authenticity of the user's identity, it prepares a PAC which described the user's privilege attributes. The Privilege service incorporates the user's PAC and a third conversation key into the PTGT, which is encrypted using the Authentication service's secret key. The PTGT envelope is encrypted using the second conversation key and also includes the third conversation key.

8. The Security client runtime decrypts the PTGT envelope using the second conversation key, and learns the third conversation key. It cannot decrypt the PTGT itself, since the PTGT is encrypted under the secret key of the Authentication Service.
9. At this point the security server has authenticated the user's identity and the user is able to acquire information about its privilege attributes that the Privilege service has certified. The client now calls sec_login

   _setup_identity( ) to set the login context to the identity that has been established. Henceforth, processes invoke by this user assume the user's login context.

3.5.3.3 Example

```
        .
        .
#include"/usr/include/csms/CsDcAuthn.h"
        .
        .
EcTInt  loginStatus;
EcTChar *principalName;
EcTChar *principalPasswd;
        .
        .
CsDcChkAuthn("ecs", "ecsguest", &loginStatus);
printf("loginStatus: %d\n", loginStatus);
        .
        .
principalName = (char *)malloc(24);
principalPasswd = (char *)malloc(24);
        .
strcpy(principalName, "ecs");
strcpy(principalPasswd, "ecsguest");
        .
        .
CsDcChkAuthn(principalName, principalPasswd, &loginStatus);
printf("loginStatus: %d\n", loginStatus);
system("klist");
        .
        .
free(principalName);
free(principalPasswd);
        .
        .
```

3.5.3.4 Files

README.authnapi:This file describes the Authentication API.

CsDcAuthn.c: This file contains the source code for the
CsDcChkAuthn( ) and other Internal functions.

CsDcAuthn.h: This file contains the declaration for the
CsDcChkAuthn API.

CsDcAuthnError.h:This file contains the status codes returned
by the CsDcChkAuthn API.

CsDcAuthnI.h:This file contains internal declarations for
the CsDcChkAuthn API.

EcTypes.h

libCsDcAuthn.make.hp:This Makefile creates the shared library for
the CsDcChkAuthn API: /usr/lib/libCsDcAuthn.sl

3.4.3.5 Build process

To build the library, type the following:

```
% make -f libCsDcAuthn.make.hp
```

This results in creation of the library: /usr/lib/libCsDcAuthn.sl (edit the make file to create the library in current directory).

Any application which invokes the Authentication API needs to be linked with the library libCsDcAuthn.sl first and also include the header files "/usr/include/csms/CsDcAuthn.h" and "/usr/include/csms/CsDcAuthnError.h".

3.5.4 User Authentication Dynamic Model

3.5.5 User Authentication Requirements

1. SCDO shall provide FOS the capability to add user account information (user name, password) to the system.
2. SCDO shall provide FOS the capability to delete user account information (user name, password) from the system.
3. SCDO shall provide FOS an API to verify a user's name and password.
4. SCDO shall provide FOS an API to request a list of FOS user roles.
5. SCDO shall maintain a list of user information (i.e., user name, user role) for all users currently logged onto the system.
6. SCDO shall provide FOS an API to add a user to the list of users currently logged onto the system.

7. SCDO shall provide FOS an API to delete a user from the list of users currently logged onto the system.
8. SCDO shall provide FOS an API to modify the user role of a user currently logged onto the system.
9. SCDO shall provide FOS an API to query information from the list of users currently logged onto the system.
10. SCDO shall provide FOS the capability to add a role to the list of FOS user roles.
11. SCDO shall provide FOS the capability to delete a role from the list of FOS user roles.
12. All the services that SCDO provides shall be available on all major platforms (Dec, Sun, HP, and SGI).

3.6 Authorization

This section contains the description of the Authorization Service, which is provided by the Communications and System Management Segment (CSMS) and is used by FOS.

3.6.1 Authorization Description

CSMS will provide FOS with an authorization service that determines if a user should be allowed to access a service or resource based on predefined criteria. The service will provide feedback to the invoking application, that will in turn allow or deny the requested access.

3.6.2 Authorization Context Within FOS

The authorization service will be used by FOS to grant privileges to EOC users. A partial list of EOC Privileges follows:

1. Command Authority - Granted to one EOC operator per spacecraft for the purpose of sending real-time commands to a specific EOS spacecraft.
2. Ground Control Authority - Granted to one EOC operator per logical string for the purpose of modifying the ground configuration for a specific EOS spacecraft .
3. IST Management Mode - Granted to IST users for the purpose of performing an IST management function. Could be use to authorize IST users for command requests. (Point-of-Contact - Jim Creegan)
4. Scheduler - Granted to EOC users for the purpose of scheduling activities. There will be different Scheduler privileges for each IST site (e.g., CERES can only modify CERES schedules). (Point-of-Contact - Bill Moore)
5. System Administrator - Grants a privilege analogous to "root" for the EOC.
6. DB Administrator - Grants a privilege analogous to "root" for EOC database manipulation.
7. IST USER - Superset of IST Management Mode, allows user login to IST and do basic functions.
8. EOC User - One who can log into EOC and do basic functions.

3.6.3 Authorization Scenario

Invoking the IsAuth function within the appServerObj to check a client's authorization privileges is accomplished as follows:

```
class EcsAclDb;

EcsAclDb *_database;

if(_database->IsAuth("EcsAppObj")
        appMethodX(...);
else
        traceobj << "Not Authorized to Perform appMethodX\n";
```

If the requesting client has permission for appMethodX, IsAuth returns TRUE, otherwise it returns FALSE.

The application developer creates serverkeytab file using the rgy_edit utility. Also when the server is still up and running, the "acl_edit" client can be run in order to manipulate ACLs. These procedures are further explained in the Release A CSMS/SDPS Internal Interface Control Document for the ECS Project (313-CD-004-001).

3.6.4 Authorization Dynamic Model

3.6.5 Authorization Requirements

1. CSMS shall provide FOS with ACLs that can be queried for information regarding user privileges associated with any resource within the network.
2. CSMS shall allow the FOS to define the domain (principals) and range (permissions) of the ACLs associated with resources. (FOS will define domain elements to be users of EOC User Stations, as well as the User Stations themselves.)
3. CSMS shall allow the FOS to specify any principal (local as well as foreign) in the domain.
4. CSMS shall provide the FOS with the GUI necessary to allow authorized FOT members to add and delete users to/from resource ACLs.
5. The authorization service will provide user interface to accept and implement updates to the ACLs associated with services, so that other system routines, as well as third party vendor products can access them.

3.7 Security Service

3.7.1 Overview

In distributed systems applications rely on services provided by servers running in different address spaces on heterogeneous platforms. Servers are independent and their main functionality is to listen for client requests, process the request and send the results back to the clients. This division of processing can be done for any number of reasons such as efficiency, data availability, etc. In addition to a client invoking a request, and the server processing that request, both the client

and the server may need to use mechanisms to protect resources as well as the integrity of the data exchanged. These mechanisms are authentication, authorization, tamper-proofing (for data integrity) and encryption (for data privacy). While authentication should always be used in every conversation between a client and a server, the mechanisms for authorization, data integrity and privacy are based on security policies of the system(s) and the application-specific need for those mechanisms. These concepts are explained in detail in the CSMS requirements document (DID 304). This document explains how these mechanisms are achieved in the current design.

Authentication is the process of verifying the validity of a principal. Authentication is usually done at two points. Initially when users login to the ECS domain, authentication is done by a "trusted third party" who supplies server's credentials for principals to use with application servers .

Authorization is the processing of deciding what sort of users/groups should be allowed to access what services/resources and then allow/deny the service. In authorization, each resource is associated with a list of permissions that should be granted to different kinds of user and different kinds of access operations. This is used to selectively to grant certain principals access to some resources. Authorization is performed by Access Control List (ACL) mechanism. An ACL is an entry with information such as the name of the user/group and the permissions list associated with them. which indicates the kind of permission/s given for the user/group. ACLs should be created and maintained for all the application specific objects.

When data is transmitted over the network from one application to another, the integrity of the data should be preserved. This is to make sure that the copy of the data the receiver gets is exactly same as the data that the sender sends. This tamper-proofing of the data may or may not be used with methods for protecting the privacy of the data. Checksums or secure hashes are often used to guarantee data integrity.

Encryption is the process of encoding a message into cipher text using a key. The process of decoding the cipher text to its original form using a key is called decryption. Encryption is used to maintain the privacy of data transmitting over the network.

While authentication is necessary, other features; authorization, checksums and data privacy are optional. ECS application can selectively choose to use these security features depending on their specific needs.

3.7.2 Context

It is assumed that all ECS services (but not necessarily all clients) run in the DCE/OODCE environment. M&O needs to be able to create and maintain accounts for principals to login to ECS hosts. The authentication service is used by the ECS Login service and may be used by both the SDPS/FOS clients and servers to authenticate each other. The authorization service may be used by any server to protect access to a resource, such as allowing only the authorized Principal Investigator to browse through some instrument data. Data integrity may be used by in IST sessions to make sure that the command requests received from Instrument Investigators are not modified in transit. Encryption may not be needed by most ECS applications directly, but is used internally by the other three security services.

3.7.3 Directives and Guidelines

The OODCE COTs product will be used for the security functionalities along with the following CSS provided custom software:

- EcsSecurity

  Provides a layer encapsulating the underlying security classes such as EcsAcl, EcsAclDb, EcsModifyableAcl, EcsAclStorageManager, DCEAclMgr, DCEAclSchema and DCESecId.

  Application developers should be able to create ACLs for application specific objects by using this class. When acls are updated and the application server goes down for some reason, when it is brought up again, the acls available are the initial acls and not the updated acls. In order to have the latest updated acls available for use by the application server every time it is brought up, this persistent storage is provided. Everytime acl is updated it is written into persistent storage file and whenever application server is brought up, the updated acls are written from the persistent storage file back into the memory for use by application server. For this functionality it is required to implement the methods of the following classes: EcsAcl, EcsModifyableAcl, EcsAclStorageManager and EcsAclDb.

- EcsFilePassword

  This inherits from the COTs provided class DCEPassword. During login context, in case of interactive principals, password is supplied on command line. The DCEPassword class takes a keyfile name as one of the arguments and provides non-interactive principals with their passwords to be stored in the keyfile, so that non-interactive principals (servers) can establish their identity without doing a login.

The server keytab file and the persistent aclfile should be placed in the same directory where the database server is going to be executed. It is required to give read permission to the server principal for the serverKeyTabFile, read and write permission for the persistentAclFile. Both these files should be placed in the same directory where the database server will be executed. The persistentAclFile will always have the latest Acls of objects.

Besides, the following COTS provided operator interfaces can be used for creating and maintaining user accounts and Acls, create server keytab file, etc. :

- Operator will use acl_edit to modify permissions to principals to access different services.
- Through rgy_edit operator will create the server keyTabFile to give an application server principal its own identity during its login context.
- Through rgy_edit, operator creates user accounts.

Section 4.2.2.4 provides examples to demonstrate why and how each of the classes and the methods should be used by an application developer. Section 4.2.2.6 gives a brief description of all the COTS and CSS provided security classes. For more information regarding the COTS utilities, refer to the DCE administration guide.

3.7.4 Sample Application Program Interface

Sample #1

Setting Client/Server Authentication Preferences

Description

For a secure communication, the first step an application developer has to take, is to set the following authentication preferences for both the client and the server objects:

Authentication Protocol - Authentication protocol can be DCE shared-secret key authentication, where the server gets its password from a keytab file for establishing its login context, or no authentication, where no tickets are exchanged, or DCE default authentication service (The current default authentication service is DCE shared-secret key.), or the DCE public-key authentication (which will be supported by DCE 1.2). This is specified by the server (per process) to indicate the type of authentication it is OFFERING, and by the client (on a per object basis) to indicate the type of authentication it DESIRES to have.

Authorization Protocol - Authorization protocol can be either No authorization, where the server performs no authorization, or Name-based, where the server performs authorization based on the client's principal name, or PAC/DCE based, where the server performs authorization using the client's DCE Privilege Attribute Certificate (PAC) sent to the server with each RPC made with *binding*. The type of authorization protocol is specified by the client to indicate the authorization type DESIRED by the client.

Protection Level - Protection level specifies the protection level for RPCs made using *binding*. It determines the degree to which authenticated communications between the client and server are protected by the authentication service specified by authentication protocol. The protection level when selected to be "packet_integrity" will ensure data integrity (i.e. ensures that data is not modified during transit) by adding encrypted checksums to the data. Also specifying the protection level as "packet_privacy" will ensure the privacy of data through the use of secret-key encryption. However in trying to achieve data integrity/privacy there is a tradeoff. i.e. more restrictive the protection level, the greater the negative impact on performance.

i. The appClientObj (**per object basis**) must invoke the *SetAuthInfo()* API as follows:

a. Include files

```
#include "appC.H"      // Generated by application IDL.
```

b. Construct an instance of the Client Class

```
appClientObj appClObj;
```

c. Invoke the SetAuthInfo API

```
appClObj.SetAuthInfo(
(unsigned_char_t*) princName,// Name to identify server principal.
rpc_c_protect_level_pkt_integ,// Specifies packet level protection which is the //    highest level guaranteed to be present in the RPC                // runtime.
```

rpc_c_authn_dce_secret, // Specifies DCE secret-key authentication protocol.

(rpc_auth_identity_handle_t)NULL, // NULL specified to use the default security login // context for the current address space.

rpc_c_authz_dce); // PAC/DCE based authorization protocol specified.

ii. The ESO (DCEServer object 'theServer', already defined in the DCEServer.C file) should invoke the SetAuthInfo() (**per process**) as follows:

a. Include files

#include <oodce/Server.H>

#include "appS.H" // Generated by application IDL.

b. Invoke SetAuthInfo

theServer->SetAuthInfo(

(unsigned_char_t*) princName,// Specifies the Principal Name to use for the Server // when authenticating RPCs.

rpc_c_authn_dce_secret,// DCE secret-key authentication protocol specified.

(void*) keyFile); // Specifies the 'KeyFile' where server gets // password.

Sample #2

Key Management and Login Context

Description

Next is the key management and login context part. The authentication mechanism is based on two fundamental constructs: *principal identities* and *secret keys.* The basic authentication policy issues therefore have to do with how applications manipulate these data: how they acquire their principal identities and how they maintain the security of the secret keys (i.e. In a network environment, when principals want to access the resources over the network, how will they provide the encrypted password, etc.).

When first invoked, a server process uses the login context of the user who invoked it as its principal identity. This may be sufficient for the application's purposes; however it may need to assume its own identity.

The server assumes its own identity by retrieving its secret key, which is analogous to a user's password, and using the key to establish its own login context. The server's key is stored in two places: by the server in a local key data file; and by the Security Service in its Registry Service database. Keys for servers that require root access to file system data or for servers that need to run as root are stored in the system-wide key file which is owned by root. Servers that do not need to run as root should store their keys in a private file, which the server has access to but nobody else does. The server's local copy is used by the server runtime to decrypt incoming client tickets, and is also by server to acquire its own login context.

The server will establish its password and login context as follows:

a. Include files

```
#include <oodce/Password.H>
#include <oodce/Login.H>
#include "EcsFilePassword.H"
```

b. Establish Server Password

(The EcsFilePassword class inherits from the base class DCEPassword provided by OODCE)

```
EcsFilePassword  passwrd(serverPrincName, serverKeytabFilename);
```

c. Establish Server Login Context

(DCEStdLoginContext is the default implementation of the DCELoginContext class defined by OODCE)

```
DCEStdLoginContext  loginCntxt(&passwrd);
```

Sample #3

Reference Monitor

Description

The ServerProg (server main program) will create a DCERefMon object. The reference monitor is used to provide mutual authentication (that is, allow the server to check that the client is as claimed) and to ensure that the server is willing to meet the client preferences for protection and authorization.

It can perform basic checks before any application code is entered. In general, these checks are as follows:

- Is the client program authenticated (i.e., has the client principal established a login context?)
- Does the protection level requested by the client meet the requirements of the server program?
- Does the authorization model requested by the client meet the requirements of the server program?

Instantiation of DCERefMon object is as below:

a. Include files

```
#include <oodce/RefMon.H>
```

b. Instantiation of the RefMon Object :

```
DCEStdRefMon*thisRefMon = new DCEStdRefMon(protectLevel, authnSvc,
authzSvc);
```

Sample #4

Generating Object UUID

Description

The ServerProg creates a DCEUuid object and initializes it with an object uuid created using uuidgen command (Refer to DCE manual for details on uuidgen) as follows:

a. Include files

```
#include <oodce/Uuid.H>
```

b. Instantiate DCEUuid object

```
DCEUuidobjectUuid("34c53cfa-9b3d-11cc-adaf-080009627155");
```

Sample #5

Initialization of AclManagement

Description

The ServerProg creates DCEAclMgr object through DefineAclMgr macro. The DCEAclMgr class allows server developers to register a rdacl interface manager object with the global DCEServer object. There is one instance of DCEAclMgr per application server, which is referred by a global reference called acl_manager. Creation of the DCEAclMgr global object is as below:

a. Include files

```
#include <oodce/AclStorageManager.H>
#include <oodce/AclMgr.H>
#include "appS.H"                    // Generated by application IDL.
```

b. Instantiate DCEAclMgr object

(Note: The DCERefMon object must be instantiated before calling the DefineAclMgr macro.)

```
DefineAclMgr(*thisRefMon, objectUuid, "database server");
```

Here the first argument DCERefMon object will enforce the security policy for requests coming through the rdacl interface. The second parameter, an object uuid will be registered with CDS and with the endpoint mapper to enable acl_edit tool to contact the right server. The third parameter is the server name used to identify this instance of a DCEAclMgr.

Sample #6

Acl Management

Description

The concept of Access Control Lists (ACLs) is used to perform authorization. DCE/ OODCE provides a set of mechanisms for access controls, which include;

- The authenticated identity and privilege attributes (in the form of credentials) of service requesters, provided by the RPC runtime to servers.
- ACLs which servers may associate with objects they control.
- A default mechanism for determining a service requester's privileges from an ACL and the requester's credentials.
- Tools for administering ACLs.

Create an instance of EcsSecurity object. Through this object invoke functions for performing CreateAclSchema(..), GetAclSchema(..), CreateAcl(..), CreateAclDatabase(..), etc. as follows:

a. Include files

```
#include <oodce/AclSchema.H>
#include <oodce/AclDb.H>
#include "EcsSecurity.H"
```

b. Instantiate EcsSecurity object

```
EcsSecurity *EcsSecurityObj = new EcsSecurity(char *databaseName);
```

c. Create acls, acl schema and acl database

Through this object can invoke the following functions:

```
EcsSecurityObj->CreateAclSchema();
ECSAclDb *EcsSecurityObj->CreateAclDatabase(databaseName, *_Schema, char *persistentDbName);
EcsSecurityObj->CreateAcls();
```

Sample #7

Check on Client's Authorization Privileges

Description

The application server is brought up and it should be running. The application client when run, will be requesting some permission to a resource (as indicated earlier a resource can also be an idl implemented method). The application server will check the ACL associated with the object/ resource and compares it with the client's PAC and makes a decision about the client's requested access to the resource. If authorized the client will perform the operation else an exception is thrown by the server on to the client's side.

Invoking the IsAuth function within the appServerObj to check client's authorization privileges:

```
class EcsAclDb;
EcsAclDb *_database;
if(_database->IsAuth("EcsAppObj")
        appMethodX(...);
else
        traceobj << "Not Authorized to Perform appMethodX\n";
```

If the requesting client has permission for *appMethodX*, IsAuth returns TRUE, otherwise it returns FALSE.

Application developer creates serverkeytab file using rgy_edit utility (explained in one of the scenarios below). Also when the server is still up and running, the "acl_edit" client can be run in order to manipulate ACLs (described in one of the below scenarios).

Sample #8

Using the COTS provided operator interface utility "rgy_edit", to create server keyfile.

Description

The application developer invoke the rgy_edit program and run the ktadd command providing the server principal name and the server keytab file name as "ktadd -p SrvPrincName -f SrvKeyFileName". The utility prompts for a password "Enter password:" twice. Supply the password both times. Exit rgy_edit utility. Give the server principal at least READ permission to this file "SrvKeyFileName" (This is important). Now the server principal is able to establish its identity during login by getting its password from this keyTabFile. For more information, refer to DCE Administration Guide.

Following is an example as to how you can run **rgy_edit** utility for creating a server keytab file:

a. Login to dce as an authenticated user.

b. Running rgy_edit Utility

i. Enter rgy_edit by typing **rgy_edit**

ii. You get **rgy_edit=>** prompt.

iii. At the above prompt type the following command to create the **srvKeyFile** :

ktadd -p srvPrincName -f srvKeyFile

iv. You get **Enter Password:** prompt to enter password

v. At the above prompt supply the **srvPrincPassword**.

vi. You get **Re-enter password to verify:** prompt for password verification

v. Supply **srvPrincPassword** again.

vi. At this stage the srvKeyFile is created. Verify if the srvKeyFile is created (OPTIONAL) with the following command:

ktlist -f srvKeyFile

vii. If the **srvKeyFile** is created, you get the following message:

/.../cellName/srvPrincName

(indicating the owner of the **srvKeyFile** created to be **srvPrincName**.)

else you get the following message:

Unable to retrieve(s) - Specified key table not found

viii. Exit out of rgy_edit utility, by typing **q** OR **e**.

ix. Provide the **srvPrincName** with at least READ permission to the **srvKeyFile**. (IMPORTANT)

NOTE: For more help refer to DCE Administration Guide and also when you are in rgy_edit type **h** for help.

Sample #9

Using COTS provided "acl_edit" utility, to manipulate Acls.

Description

M&O staff invoke the acl_edit program and then identify a resource (the access method associated with a database server application) to view the ACL associated with the resource. The acl_edit program calls the database server program to get a printable representation of the ACL associated with the access operation. The database servers view operation checks if the client can perform the view operation and returns the ACL associated with the access operation in a printable format. Acl_edit then displays this on the screen. The M&O staff then invoke the edit operation to edit the selected ACL. The acl_edit invokes the read acl operation to get a copy of the acl associated with the access operation. Upon checking the permissions associated with the read operation, a copy of the ACL is sent back to the acl_edit. The M&O staff change this ACL and invoke the save operation. The acl_edit then invokes the replace acl operation of the database server passing this modified ACL. The replace operation after checking the authorization privileges, replaces the supplied ACL in memory and may update the contents onto the disk.

Following is an example as to how you can run **acl_edit** tool to manipulate ACLs:

a. Login to dce as an authenticated user with proper privileges (ex: privilege to modify an object's ACL)

b. Run the application (which manages the object/s for which you want to run acl_edit for) server executable.

c. Run **acl_edit** for object (say) **obj1**

i.Type the following command:

acl_edit /.:/cdsDirName/appSrvCdsName/obj1

ii. You get **sec_acl_edit>** prompt

iii. View the object's ACL list by typing **l**. You see the ACL similar to the following example:

 # SEC_ACL for /.:/cdsDirName/appSrvCdsName/obj1

 # Default cell = /.../edfcell.hitc.com

unauthenticated:r

group:CSS:rtx

user:manand:rwcda

any_other:rt

iv. Modify ACL. Following are some examples.

Example 1 :m user:joe:rwta

(adding a new entry for user joe in the above shown example)

Example 2:m group:CSS:rtxa

(modifying the ACL entry for group CSS users)

v. Commit the above modifications by typing **co**

vi. View the modified ACL list. It will now look as follows:

 # SEC_ACL for /.:/cdsDirName/appSrvCdsName/obj1

 # Default cell = /.../edfcell.hitc.com

unauthenticated:r

group:CSS:rtxa

user:manand:rwcda

user:joe:rwta

any_other:rt

vii. Exit out of acl_edit by typing **e**.

NOTE: For more information on acl_edit utility, refer to DCE reference manuals and type **h** (for help) when you are in acl_edit.

3.7.5 Object Model

A brief description of all the CSS customized security objects as well as the COTS provided classes used by the CSS security is provided in the Table 3.7.5-1 below. For more information on any of the COTS provided objects, please refer to the OODCE reference manuals. CSS customized security classes are discussed in detail in the Release A CSMS Communications Subsystem Design Specification for the ECS Project, Section 3.7

Table 3.7.5-1. Security Object Responsibility Matrix (1 of 2)

Class	Description
rgy_edit	This is command line interface used to create DCE accounts for principals, create keytab files for non-interactive principal (servers) to maintain its password, etc.
acl_edit	This is command line interface to manipulate (delete, insert, read, write, etc.) ACLs associated with objects.
DCEObj	This is a base class that stores information about the DCE interfaces implemented by a concrete manager class.
DCEUuid	This is the utility class that encapsulates the DCE data type uuid_t.
DCEInterface	This is a base class that encapsulates basic functionality for client objects.
DCEInterfaceMgr	This is a base class that encapsulates the functionality common to a DCE interface manager.
ESO	This is the Global Server Object. It manipulates manager objects and interacts with the DCE subsystems.
DCERefMon	This class provides an abstraction of a reference monitor that controls the client object's access to a manager object. By deriving from this abstract base class, various reference monitors that provide different security policies can be implemented.
DCEAclMgr	This class registers a rdacl interface manager object with the global DCEServer(ESO) object. DefineAclMgr macro is used to construct this class.

Table 3.7.5-1. Security Object Responsibility Matrix (2 of 2)

Class	Description
DCESecId	This is a utility class that encapsulates the sec_id_t data type.
DCEAclSchema	This is a class defining the ACL permission bits and print strings.
DCESchemaBitset	This class represents a set of permissions formatted according to a schema. Since most uses of ACL will use less than 32 bits of permissions, there is an efficient encoding of 32-bit ACLs.
EcsAcl	This class is used for accessing a DCE access control list. It maintains all the information about an ACL.
EcsAclDb	This is a class for ACL database. It defines the interface to the ACL database.
EcsModifyableAcl	This is a temporary copy of a EcsAcl that can be used for editing directly by the server or through an application-defined interface.
EcsAclStorageManager	This is a class that maintains a table of known ACL databases. It manages the ACL databases being used by a server, providing registration and search services for these databases.
EcsFilePassword	Provides an interface to the storage and access of password of principal (usually a server) to help the principal establish it's identity during login context.
EcsSecurity	Provides higher level functionality to authenticate principals accessing resources. These include create/update/delete ACLs, define permissions sets, persistence to the ACL database.

3.7.6 Dynamic Model Scenarios

3.7.6.1 Scenario #1

Abstract

- This is the entire security scenario which traces the interactions of the application (which implements security) client and server objects with the global server object ESO, the ECSSecurity, ECSFilePassword, etc. The following occurs in this scenario:
  - Setting client and server authentication information with RPC runtime.
  - Initialization aspects such as creating instances of different objects and registering them.
  - Creation of ACLs, AclSchema and AclDatabase.
  - Invoking of IsAuth to check client's Authorization Privileges.

Interfaces

- CSS's ECSSecurity and ECSFilePassword.

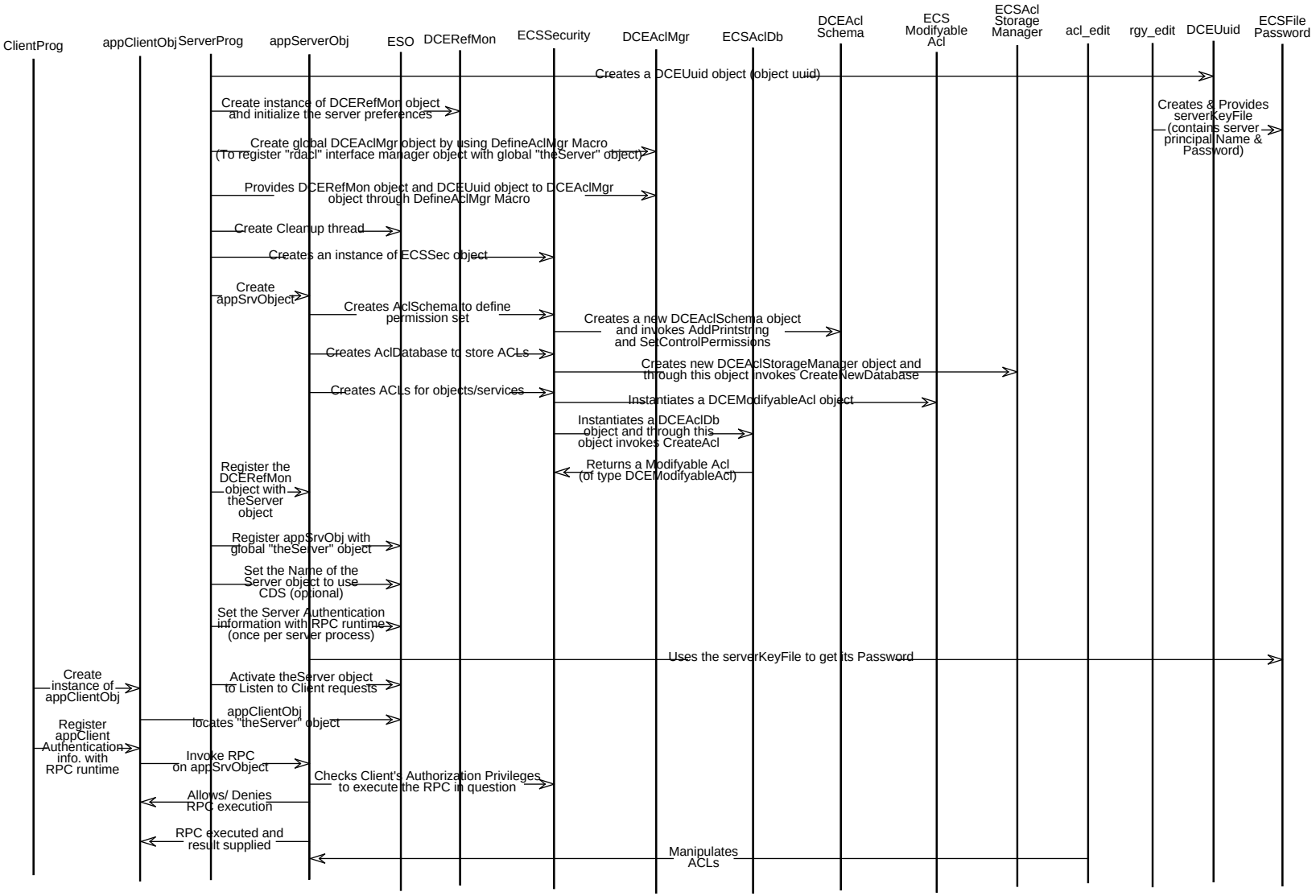

Figure 3.7.6-1. Security Event Trace #1

Stimulus

- To create and maintain access control lists(ACLs) for resources, and with the help of these ACLs perform security check to find out if a caller is authorized to access specified resource.

Desired Response

- The database server should have modifiable ACL/s for object/s which consists of ACL entries that allow a permission set defined by ACL schema, an ACL database to store the ACL created. Functionality to check caller's privilege to access any desired resource.

Participating Classes

- CSS's ECSSecurity and ECSFilePassword classes.
- COTS provided classes: DCEAclMgr, DCERefMon, ECSAcl, ECSAclDb, etc.

Pre-conditions

- Segments implementing security need to run in the DCE environment.

Post-conditions

- The database client and server authentication information are registered with their RPC runtime. A modifiable ACL with at least one person granted with the privilege to change ACL as desired, an ACL schema with default 7 permissions (read, write, execute, test, insert, delete, acl_control) and a provision to define 25 application specific permissions, an ACL database which has the latest updated ACL are available to the application server implementing security. The rdacl interface manager object is registered with theServer global object to provide the functionality to use MSS provided acl_edit utility to manipulate ACLs. A function to check client's authorization privileges to access resources is available.

Scenario description

- For a secure communication, the first step an application developer has to take, is to set the following authentication preferences for both the client and the server objects:

  Authentication Protocol - Authentication protocol can be DCE shared-secret key authentication, where the server gets its password from a keytab file for establishing its login context, or no authentication, where no tickets are exchanged, or DCE default authentication service (The current default authentication service is DCE shared-secret key.), or the DCE public-key authentication (which will be supported by DCE 1.2). This is specified by the server (per process) to indicate the type of authentication it is OFFERING, and by the client (on a per object basis) to indicate the type of authentication it DESIRES to have.

  Authorization Protocol - Authorization protocol can be either No authorization, where the server performs no authorization, or Name-based, where the server performs authorization based on the client's principal name, or PAC/DCE based, where the server performs authorization using the client's DCE Privilege Attribute Certificate (PAC) sent to the server with each RPC made with *binding*. The type of authorization protocol is specified by the client to indicate the authorization type DESIRED by the client.

Protection Level - Protection level specifies the protection level for RPCs made using *binding*. It determines the degree to which authenticated communications between the client and server are protected by the authentication service specified by authentication protocol. The protection level when selected to be "packet_integrity" will ensure data integrity (i.e. ensures that data is not modified during transit) by adding encrypted checksums to the data. Also specifying the protection level as "packet_privacy" will ensure the privacy of data through the use of secret-key encryption. However in trying to achieve data integrity/privacy there is a tradeoff. i.e. more restrictive the protection level, the greater the negative impact on performance.

i. The appClientObj (per object basis) must invoke the *SetAuthInfo()* API as follows:

a. Include files

```
#include "appC.H"      // Generated by application IDL.
```

b. Construct an instance of the Client Class

```
appClientObj appClObj;
```

c. Invoke the SetAuthInfo API

```
appClObj.SetAuthInfo(
(unsigned_char_t*) princName,// Name to identify server principal.
rpc_c_protect_level_pkt_integ,// Specifies packet level protection which is the //    highest level guaranteed to be present in the RPC                // runtime.
rpc_c_authn_dce_secret,   // Specifies DCE secret-key authentication protocol.
(rpc_auth_identity_handle_t)NULL, // NULL specified to use the default security login // context for the current address space.
rpc_c_authz_dce);      // PAC/DCE based authorization protocol specified.
```

ii. The ESO (DCEServer object 'theServer', already defined in the DCEServer.C file) should invoke the SetAuthInfo() as follows:

a. Include files

```
#include <oodce/Server.H>
#include "appS.H"      // Generated by application IDL.
```

b. Invoke SetAuthInfo

```
theServer->SetAuthInfo(
(unsigned_char_t*) princName,// Specifies the Principal Name to use for the Server // when authenticating RPCs.
rpc_c_authn_dce_secret,// DCE secret-key authentication protocol specified.
(void*) keyFile);        // Specifies the 'KeyFile' where server gets                //
password.
```

2. Next is the Key management and login context part. The authentication mechanism is based on two fundamental constructs: *principal identities* and *secret keys*. The basic authentication policy issues therefore have to do with how applications manipulate these data: how they acquire their

principal identities and how they maintain the security of the secret keys (i.e. In a network environment, when principals want to access the resources over the network, how will they provide the encrypted password, etc.).

When first invoked, a server process uses the login context of the user who invoked it as its principal identity. This may be sufficient for the application's purposes; however it may need to assume its own identity.

The server assumes its own identity by retrieving its secret key, which is analogous to a user's password, and using the key to establish its own login context. The server's key is stored in two places: by the server in a local key data file; and by the Security Service in its Registry Service database. Keys for servers that require root access to file system data or for servers that need to run as root are stored in the system-wide key file which is owned by root. Servers that do not need to run as root should store their keys in a private file, which the server has access to but nobody else does. The server's local copy is used by the server runtime to decrypt incoming client tickets, and is also by server to acquire its own login context.

The server will establish its password and login context as follows:

a. Include files

```
#include <oodce/Password.H>
#include <oodce/Login.H>
#include "ECSFilePassword.H"
```

b. Establish Server Password

(The ECSFilePassword class inherits from the base class DCEPassword provided by OODCE)

```
ECSFilePassword  passwrd(serverPrincName, serverKeytabFilename);
```

c. Establish Server Login Context

(DCEStdLoginContext is the default implementation of the DCELoginContext class defined by OODCE)

```
DCEStdLoginContext  loginCntxt(&passwrd);
```

3. The ServerProg (server main program) will create a DCERefMon object. The reference monitor is used to provide mutual authentication (that is, allow the server to check that the client is as claimed) and to ensure that the server is willing to meet the client preferences for protection and authorization.

It can perform basic checks before any application code is entered. In general, these checks are as follows:

- Is the client program authenticated (i.e., has the client principal established a login context?)
- Does the protection level requested by the client meet the requirements of the server program?
- Does the authorization model requested by the client meet the requirements of the server program?

Instantiation of DCERefMon object is as below:

a. Include files

```
#include <oodce/RefMon.H>
```

b. Instantiation of the RefMon Object :

```
DCEStdRefMon*thisRefMon = new DCEStdRefMon(protectLevel, authnSvc,
authzSvc);
```

4. The ServerProg creates a DCEUuid object and initializes it with an object uuid created using uuidgen command (Refer to DCE manual for details on uuidgen) as follows:

a. Include files

```
#include <oodce/Uuid.H>
```

b. Instantiate DCEUuid object

```
DCEUuidobjectUuid("34c53cfa-9b3d-11cc-adaf-080009627155");
```

5. Initialization of AclManagement :

The ServerProg creates DCEAclMgr object through DefineAclMgr macro. The DCEAclMgr class allows server developers to register a rdacl interface manager object with the global DCEServer object. There is one instance of DCEAclMgr per application server, which is referred by a global reference called acl_manager. Creation of the DCEAclMgr global object is as below:

a. Include files

```
#include <oodce/AclStorageManager.H>
#include <oodce/AclMgr.H>
#include "appS.H"                    // Generated by application IDL.
```

b. Instantiate DCEAclMgr object

(Note: The DCERefMon object must be instantiated before calling the DefineAclMgr macro.)

```
DefineAclMgr(*thisRefMon, objectUuid, "database server");
```

Here the first argument DCERefMon object will enforce the security policy for requests coming through the rdacl interface. The second parameter, an object uuid will be registered with CDS and with the endpoint mapper to enable acl_edit tool to contact the right server. The third parameter is the server name used to identify this instance of a DCEAclMgr.

6. Acl Management

The concept of Access Control Lists (ACLs) is used to perform authorization. DCE/ OODCE provides a set of mechanisms for access controls, which include;

- The authenticated identity and privilege attributes (in the form of credentials) of service requesters, provided by the RPC runtime to servers.
- ACLs which servers may associate with objects they control.
- A default mechanism for determining a service requester's privileges from an ACL and the requester's credentials.
- Tools for administering ACLs.

Create an instance of ECSSecurity object. Through this object invoke functions for performing CreateAclSchema(..), GetAclSchema(..), CreateAcl(..), CreateAclDatabase(..), etc. as follows:

a. Include files

```
#include <oodce/AclSchema.H>
#include <oodce/AclDb.H>
#include "ECSSecurity.H"
```

b. Instantiate ECSSecurity object

```
ECSSecurity *ecsSecurityObj = new ECSSecurity(char *databaseName);
```

c. Create acls, aclschema and acl database

Through this object can invoke the following functions:

```
ecsSecurityObj->CreateAclSchema();
DCEAclSchema *ecsSecurity-> GetSchema();
ECSAclDb *ecsSecurityObj->CreateDatabase(databaseName, *_Schema, char *persistentDbName);
```

d. The application server is brought up and it should be running. The application client when run, will be requesting some permission to a resource (as indicated earlier a resource can also be an idl implemented method). The application server will check the ACL associated with the object/ resource and compares it with the client's PAC and makes a decision about the client's requested access to the resource. If authorized the client will perform the operation else an exception is thrown by the server on to the client's side.

Invoking the IsAuth function within the appServerObj to check client's authorization privileges:

```
class ECSAclDb;
ECSAclDb *_database;
if(_database->IsAuth("ecsappobj")
        AppMethodX(...);
else
        traceobj << "Not Authorized to Perform AppMethodX\n";
```

If the requesting client has permission for *AppMethodX*, IsAuth returns TRUE, otherwise it returns FALSE.

Application developer creates serverkeytab file using rgy_edit utility (explained in one of the scenarios below). Also when the server is still up and running, the "acl_edit" client can be run in order to manipulate ACLs (described in one of the below scenarios).

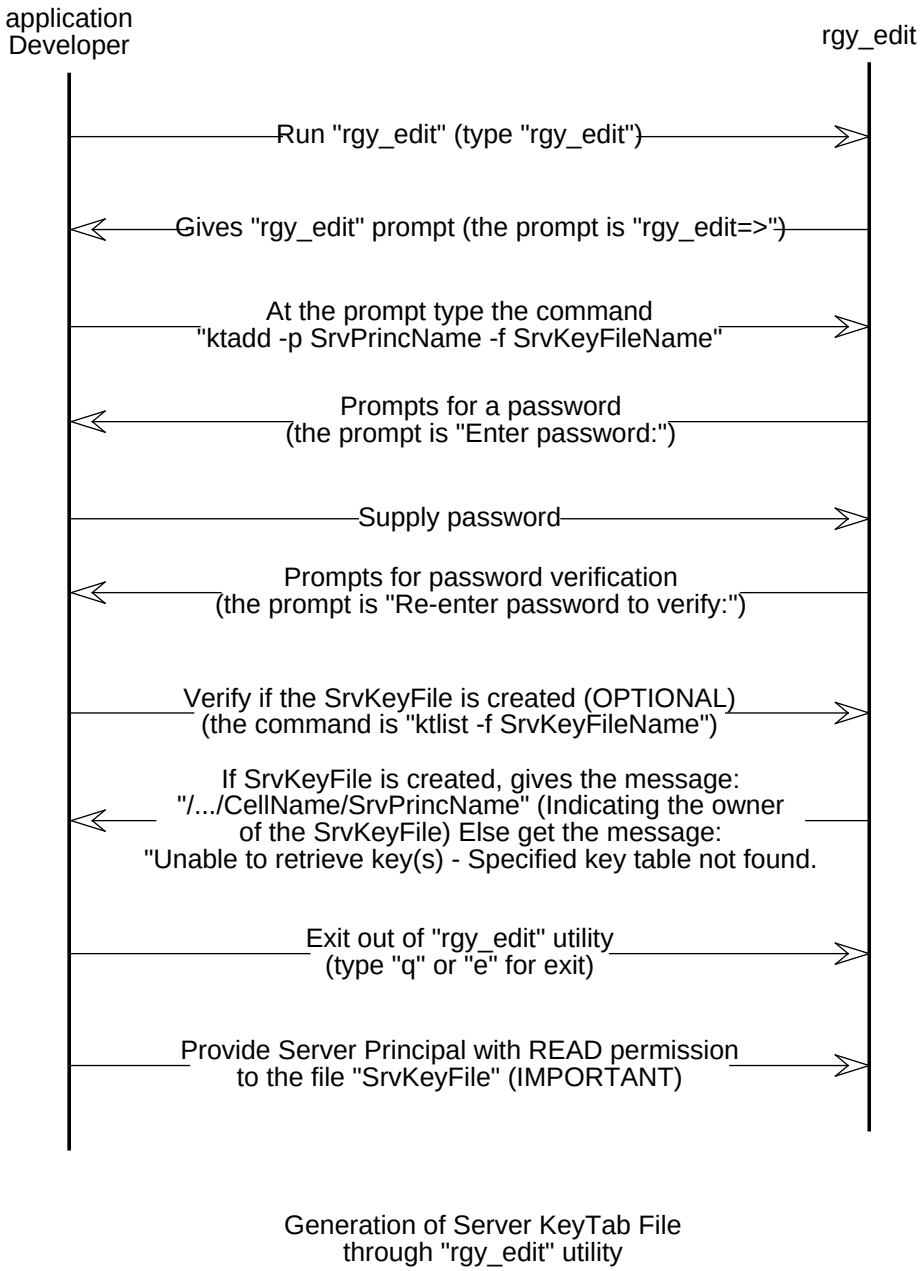

Figure 3.7.6-2. Security Event Trace #2

3.7.6.2 Scenario #2

Abstract

- This scenario traces the interactions of the application developer with the DCE/OODCE provided "rgy_edit" utility, to create the server keytab file to store the server principal's (principal under who's identity the server is to run) password.

Interfaces

- COTS provided rgy_edit application.

Stimulus

- Application developer (dce authenticated user) invoke the rgy_edit function to create the server keytab file.

Desired Response

- The server keytabfile should be created and available for use by the database server to get its password to establish it's identity during login context.

Participating Classes

- rgy_edit

Pre-conditions

- The application developer should be a DCE authenticated user in order to create the keytab file through "rgy_edit".

Post-conditions

- Then serverKeyTabFile is available for use by the database server to establish its identity during login context..

Scenario description

- The application developer invoke the rgy_edit program and run the ktadd command providing the server principal name and the server keytab file name as "ktadd -p SrvPrincName -f SrvKeyFileName". The utility prompts for a password "Enter password:" twice. Supply the password both times. Exit rgy_edit utility. Give the server principal at least READ permission to this file "SrvKeyFileName" (This is important). Now the server principal is able to establish its identity during login by getting its password from this keyTabFile. For more information, please refer to DCE Administration Guide.

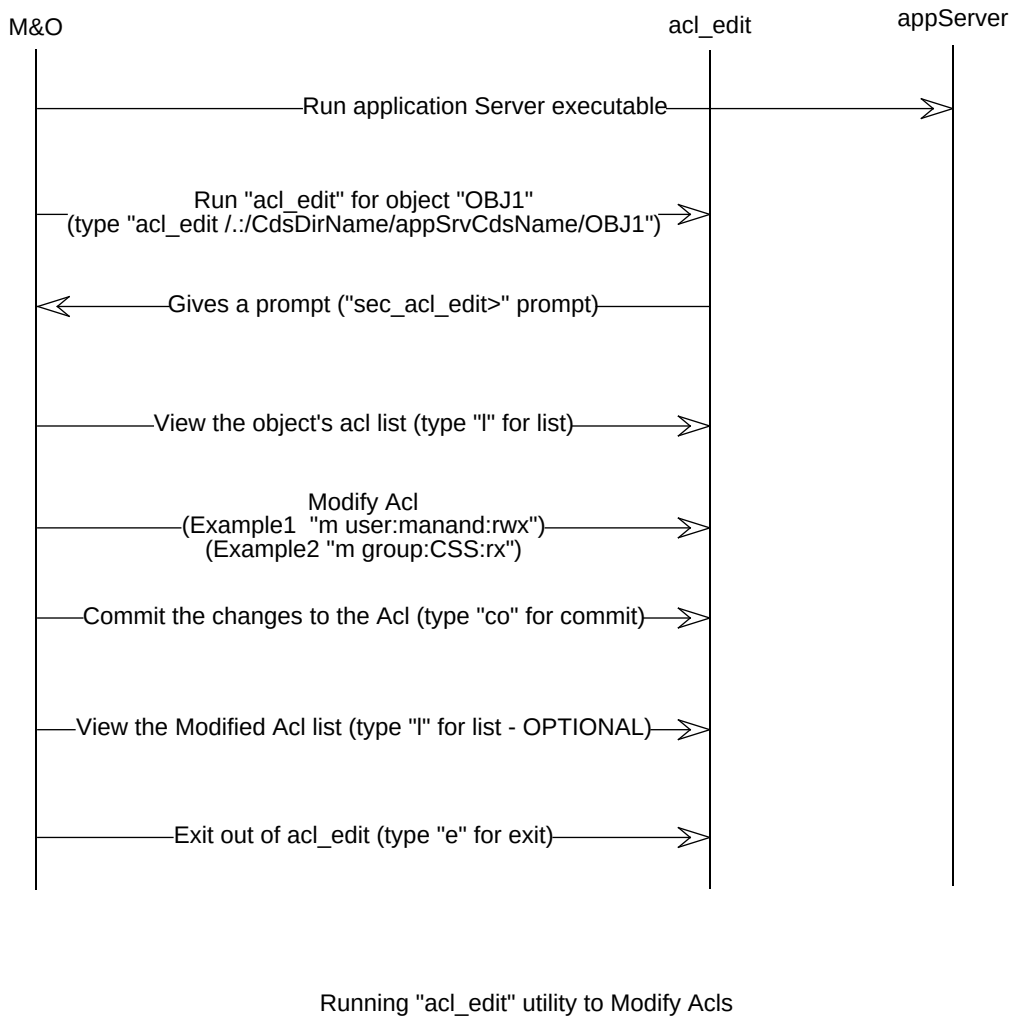

Figure 3.7.6.-3. Security Event Trace #3

3.7.6.3 Scenario #3

Abstract

- This scenario traces the interactions of M&O staff modifying the contents of an ACL associated with an access operation of a database server through an MSS-provided interface.

Interfaces

- MSS's acl_edit application.

Stimulus

- M&O staff invoke the acl_edit function to edit the ACL associated with given resource.

Desired Response

- The server should use the modified ACL in any future authorization checks performed in deciding whether a given client can access the resource associated with the ACL.

Participating Classes

- acl_edit

Pre-conditions

- The database server is running and has implemented the rdacl calls associated with the database server.

Post-conditions

- The newly modified ACL will be attached to the resource.

Scenario description

- M&O staff invoke the acl_edit program and then identify a resource (the access method associated with a database server application) to view the ACL associated with the resource. The acl_edit program calls the database server program to get a printable representation of the ACL associated with the access operation. The database servers view operation checks if the client can perform the view operation and returns the ACL associated with the access operation in a printable format. Acl_edit then displays this on the screen. The M&O staff then invoke the edit operation to edit the selected ACL. The acl_edit invokes the read acl operation to get a copy of the acl associated with the access operation. Upon checking the permissions associated with the read operation, a copy of the ACL is sent back to the acl_edit. The M&O staff change this ACL and invoke the save operation. The acl_edit then invokes the replace acl operation of the database server passing this modified ACL. The replace operation after checking the authorization privileges, replaces the supplied ACL in memory and may update the contents onto the disk.

3.7.7 Implementation

The OODCE COTs product will be used for the security functionalities defined in this section. However, CSS will provide the following custom software:

- ECSSecurity
  - Provides a layer encapsulating the underlying security classes such as ECSAcl, ECSAclDb, ECSModifyableAcl, ECSAclStorageManager, DCEAclMgr, DCEAclSchema and DCESecId.
  - When acls are updated and the application server goes down for some reason, when it is brought up again, the acls available are the initial acls and not the updated acls. In order to have the latest updated acls available for use by the application server every

time it is brought up, this persistent storage is provided. Everytime acl is updated it is written into persistent storage file and whenever application server is brought up, the updated acls are written from the persistent storage file back into the memory for use by application server. For this functionality it is required to implement the methods of the following classes: ECSAcl, ECSModifyableAcl, ECSAclStorageManager and ECSAclDb.

- ECSPassword
  - This inherits from the COTs provided class DCEPassword. During login context, in case of interactive principals, password is supplied on command line. The DCEPassword class takes a keyfile name as one of the arguments and provides non-interactive principals with their passwords to be stored in the keyfile, so that non-interactive principals (servers) can establish their identity without doing a login.

This page intentionally left blank.

4. MSS Services

4.1 MSS Services

4.1.1 MSS Overview

The MSS provides ECS Maintenance and Operations (M&O) Staff with the capability to manage the ECS enterprise, i.e., to perform network and system management services for all ECS resources, including all FOS, SDPS, and CSMS components. The MSS is composed of a combination of Commercial Off The Shelf (COTS) and custom management applications to provide a highly automated means for monitoring and managing the various ECS resources. This section describes the use of MSS to manage ECS resources, particularly FOS resources, located at the EOS Operations Center (EOC).

The Management Subsystem (MSS) provides enterprise management (network and system management) for all ECS resources: commercial hardware (including computers, peripherals, and network routing devices), commercial software, and custom applications. Consistent with current trends in industry, the MSS manages both EOC network resources and EOC host/application resources. Additionally MSS also supports many requirements allocated to FOS for management data collection and analysis/distribution.

The MSS allocates services to both the system-wide and local levels. With few exceptions, the management services are fully decentralized, no single point of failure exists which would preclude user access. In principle every service is distributed unless there is an overriding reason for it to be centralized. MSS has two key specializations: Enterprise Monitor and Coordination Services and Local System Management Services. The distribution of these services provides maximum flexibility and policy neutrality in the design and implementation of MSS services.

The MSS is composed of a variety of management applications providing services such as fault, performance, security, and accountability management for EOC networks, hosts, as well as FOS applications. The management applications reside on an MSS Server at the EOC. The management information for remote objects needs to be conveyed to the management applications through the Management Agent Service which primarily resides on remote hosts.

4.1.2 MSS Context

As shown in the FOS-MSS Context Diagram (Figure 4.1-1), the Management Agent Service provides the primary interface point between FOS subsystems and the MSS. This interface is used to communicate management requests and responses between management applications and managed resources. Fault, performance, accountability, and security management services are provided through this mechanism for all managed resources. There are other MSS services at the EOC, such as configuration management, trouble ticketing, and some aspects of security and accountability management. These other services, however, do not directly interface with FOS applications, although they may involve interfaces with FOS operators. A further description of these other services is provided in Sections 4.5 and 4.6.

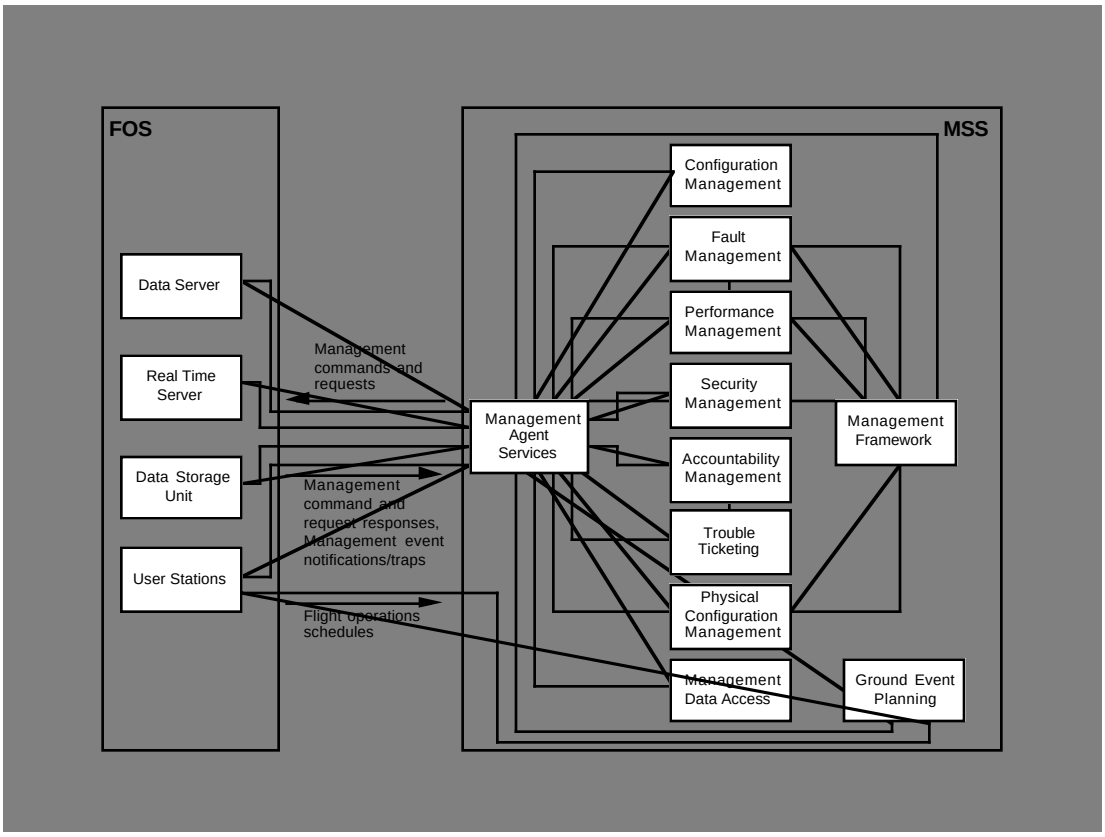

Figure 4.1-1. FOS-MSS Context Diagram

Since Management Agent Services form the basis for FOS systems management, a brief, high-level description of the agent capabilities are described here. A detailed description of some of the metrics used and their implementations are provided in the Management Services and Performance sections (4.2 and 4.3, respectively).

The MSS Management Agent Service provides the following capabilities for the management of FOS resources at the EOC:

- Can manage network devices, host systems, and FOS applications.

  Instrumentation APIs provided by MSS, are used to manage ECS-developed FOS applications. FOS application developers can determine the performance metrics that should be monitored, instrument their applications appropriately to allow the agent to monitor that metric, and identify suggested metric thresholds which can be used to detect degraded application performance.

Proxy agent(s) are used to manage non-ECS developed FOS applications (i.e., applications for which FOS developers cannot instrument their applications with the MSS Instrumentation API). Proxy agents are generally supplied by the resource provider in the case of COTS applications. The front-end of the proxy agent uses the instrumentation API provided by MSS. The back-end of the proxy agent is the interface unique to each managed application.

- Defines the managed object model to represent the management characteristics, or metrics, of ECS applications. The metrics are defined in a Management Information Base (MIB).

  MSS manages objects on each managed host using metrics defined in industry standard MIBs. In addition, MSS has defined a managed object model for ECS applications in standard MIB format. The Management Agent Service implements this application MIB, which can contain application-specific metrics, such as the length of time to complete an operation. Information contained in the MIB is composed of several different types of attributes: configuration, performance, fault, dynamic, static, and traps.

- Sends ECS management event notifications/traps to the MSS Management Framework.

  Event handling is provided by Management Agent Service to satisfy the need to dispatch events for orderly and prompt resolution to fix problems. All events are logged locally on each host. The agent can also be configured to forward certain events immediately to the management server, which can then perform a predefined action in response to the detected event. The use of the agent in collecting management events is further described in section 4.2.

- Retrieves ECS managed object metrics in response to requests from MSS applications.

  Performance data is also logged by Management Agent Services, with measured values compared against configurable thresholds. The agent can then generate event notifications of its own whenever the defined thresholds are exceeded. The use of the agent in collecting performance data is further explained in Section 4.3.

- Performs local polling on hosts to monitor the state of managed ECS resources.

  In order to reduce network traffic, the agent is capable of monitoring ECS resources remotely. The agent logs performance data, performs threshold checks, generates management events when thresholds are exceeded, logs management events/ notifications, and selectively forwards high priority management events to the Management Framework in real time. The logs are then transferred to the management server during periods of low traffic for further analysis by management applications.

- Receives ECS management Set requests from the Management Framework.

  The management agent provides the capability to send commands to managed applications through the use of set commands. This capability is further described in the Management Services section (Section 4.2).

MSS establishes an ECS file system on each managed host to store configuration-related files for applications. The root of this ECS file system is ECS. It has three subdirectories, FOS, SDPS, and CSMS. Each of these can be further subdivided into subdirectories. Every ECS application package should have its own subdirectory.

The full path name of the executable file needs to be stored in the application's directory. A file with suffix .inp is used for the file containing the required input arguments. The .pwd is used as a suffix for the keytab file. A .cfg file is created at installation time to contain configuration-related information. When an application starts, a .ins file is created for that particular application instance. It will then be deleted when the application terminates.

4.1.1.1 Management Views

At each ECS installation, including the EOC, M&O Staff autonomously provide local management services associated with its ECS resources and, hence, are provided a local management view. At the System Monitoring and Coordination Center (SMC), M&O Staff provide enterprise monitoring and coordination services associated with the all ECS installations and are provided a system-wide management view. Extensive configurability is provided by the MSS applications to enable these views to be shared or controlled as necessary based on ECS management policy. Beside providing these views to M&O Staff for monitoring and control purposes, the management services make use of legacy CSS services such as electronic mail and bulletin board for coordination. The services provided by CSMS at the SMC, located at Goddard Space Flight Center (GSFC), are collectively referred to as Enterprise Monitoring and Coordination (EMC). In the same context, services provided by CSMS at the EOC are collectively referred to as Local System Management (LSM).

ECS management at the EOC consists of two different views: the FOS service level view and the MSS site level management (LSM) view. In addition, the ECS level view at the SMC encompasses the management of the entire system, including the EOC.

As shown in Figure 4.1-2, MSS receives both FOS service-level data (such as errors, transactions, FOS service-related performance data, and other FOS-unique data) and resource data (such as resource faults, performance data, security events, and accountability events) from managed resources at the EOC. This data is gathered on each managed object at the EOC via the management agent services. MSS applications then correlate, process, and make available the gathered management data.

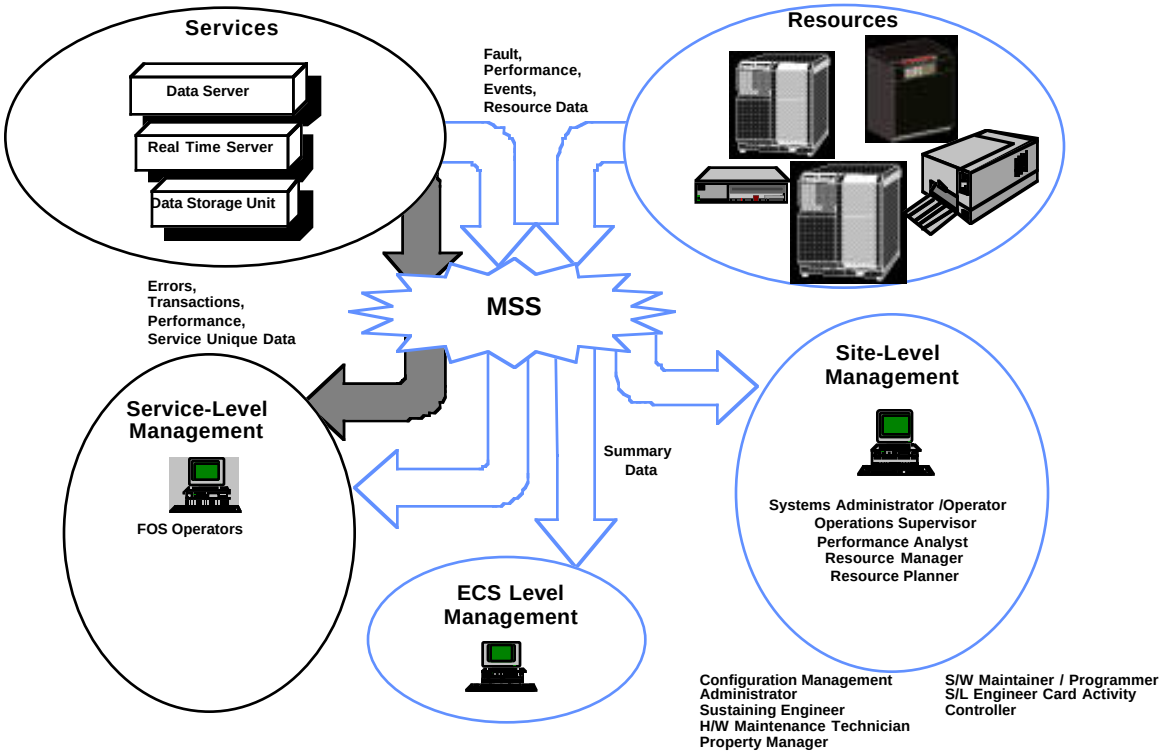

Figure 4.1-2. Management Data Flows

Table 4.1-1 provides an overview of the types of management information available from the EOC to the appropriate operators (note that the SMC operator is located separately from the EOC).

Table 4.1-1. EOC Management Data Types

Operator	Management Data	Purpose of Data
FOS Operator	FOS Service-Level Data	Provides information on the performance of FOS services.
FOS Operator	FOS Resource Data	Provides information on the status and performance of FOS resources at the EOC.
EOC MSS Operator	EOC Resource Data	Provides information on the status and performance of all ECS resources at the EOC.
SMC Operator	EOC Summary Resource Data	Provides a summarized view of the status and performance of all ECS resources at the EOC.

4.1.1.1.1 FOS Service Level View

The FOS service level view provides the FOS operator with the management information necessary to determine the overall performance of FOS systems. At the FOS service level view, FOS operators are responsible for:

- Monitoring the FOS service level performance
- Managing FOS functions
- Managing FOS tasks and processes

4.1.1.1.2 LSM View

The LSM view provides the MSS M&O operator(s) with management information on the status and performance of all managed resources at the EOC. At the LSM view, M&O operators are responsible for:

- Monitoring and management of EOC site resources
- Performing system administration functions
- Maintaining ECS user (including operator) information
- Maintaining the site software and hardware configuration
- Coordinating system maintenance activities
- Recording and tracking problem reports through resolution
- Providing summary management data to the SMC

4.1.1.1.3 SMC View

Although located remotely from the EOC, the SMC operator still has a variety of responsibilities associated with the management of EOC resources. At the SMC, M&O operators with the SMC view are responsible for:

- Administering ECS-wide planning and policy
- Monitoring "Rolled-Up" site level performance

- Monitoring the Wide Area Network
- Exchanging information with external systems
- Analyzing fault and performance trends

4.1.2 MSS Hardware

The MSS hardware at the EOC consists of the following components:

- Local System Management Servers
- Management Workstations
- Printers

4.1.2.1 Connectivity

The EOC hardware connectivity is depicted in Figure 4.1-3. The EOC LSM resides on a separate FDDI ring, with connectivity to FOS systems and the rest of the site provided by a redundant FDDI switch/router. The MSS Local Management Server is equipped with a DAS (dual-attached station) card that is connected to two FDDI concentrators, providing redundancy in the event of a concentrator failure. The MSS monitoring workstations are equipped with an SAS (single-attached station) interface card, connected to a single FDDI concentrator.

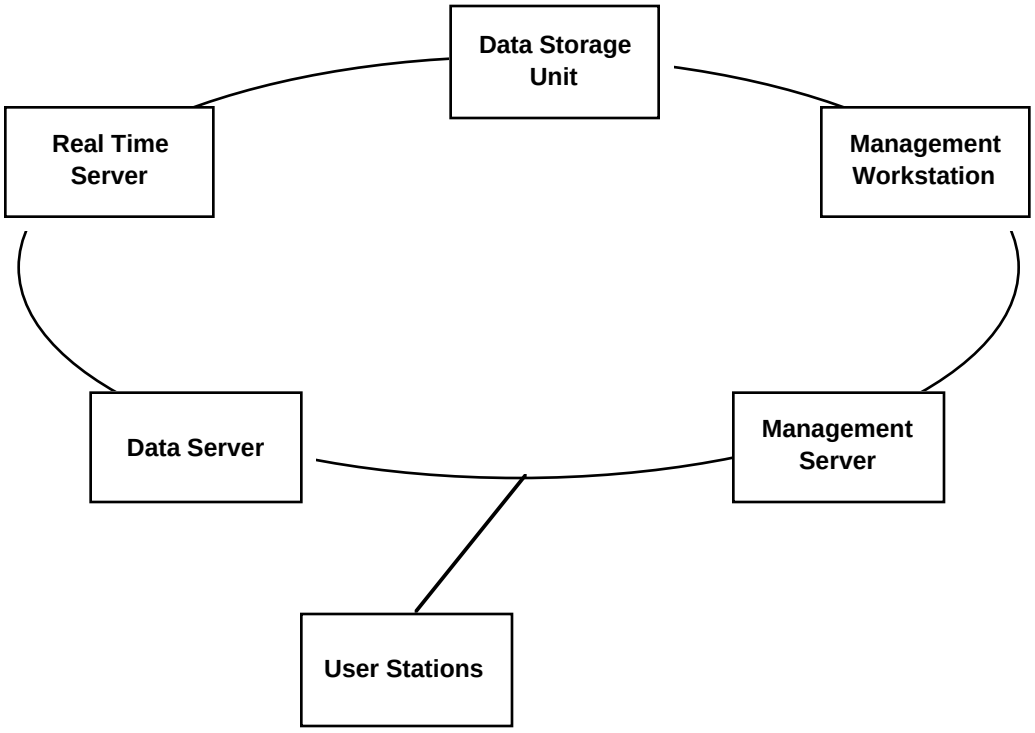

Figure 4.1-3. EOC Hardware Connectivity

4.1.2.2 MSS Hardware Components

The MSS monitoring / management server is the primary server for MSS applications and data. It is cross-strapped to the CSS communications server to provide for failover (warm standby) capability, and is populated with management applications (e.g., HP OpenView, ClearCase), common management services (e.g., Sybase, word-processing, and spreadsheet packages), and management agent services software as well as CSS client software.

The management workstation configurations are networked workstations that support all aspects of enterprise management between the M&O staff and the LSM. The management workstations are populated with the CSS client, the MSS management agent services, and user-selected subsets of the enterprise monitoring configuration software and data.

4.1.2.3 Failover and Recovery strategy

Analysis of failover strategies supports the integration of the CSS and MSS servers to serve as warm standby to each other, cross-strapped to RAID devices for critical data access by either server. MSS logical server functions are be configured but inactive on the CSS server. In the event of a failure of either server, the second RAID can be mounted for use by the backup server. All data is replicated, and is also routinely safestored in the ECS data server archive.

The LSM is designed to continue to function in the event of an EMC failure, and agents at hosts will continue to monitor managed objects in the event of an LSM failure. Dual attached FDDI within the local EOC LAN designs for critical RMA links.

Specific calculations of reliability and availability of MSS components are provided in 516-CD-001-003, Reliability Predictions for the ECS Project (August 1995), and 518-CD-001-003, Maintainability Predictions for the ECS Project (August 1995).

4.2 Management Service

4.2.1 Management Service Description

MSS provides a set of management services for use by other ECS subsystems. The following management services are provided by MSS and utilized by the FOS:

- Event logging and real-time notification

  This service provides the capability for ECS applications to log events that generate management data to a local history log file. Further, based on the type and the severity of the events, the service provides the capability to send real-time notifications to the LSM.

- ECS Resource Monitoring

  MSS Lifecycle services provide the capability for MSS to effect the discovery (registration), startup and shutdown of (long-lived or permanent) ECS applications. A separate service provides the capabilty for ECS subsystems to register and cancel requests for the monitoring of specific transient processes, with the capability to specify the identity of an interested ECS application to receive a notification in case the specified transient process fails between the initial registration and the cancellation. Together these services monitor EOC software applications for changes in status.

- Sending of notifications from MSS management applications to other ECS applications

This service provides the MSS management applications the capability to send notifications to ECS applications.

- Instrumentation of ECS applications

This service provides MSS management applications the capability to retrieve application-specific data (metrics) from ECS applications.

These management services are provided by MSS by means of a set of public classes made available to ECS subsystems. These public classes provide the interface for ECS subsystems to MSS and vice versa. These public classes, their attributes and their methods are described in Section 4.2.3 of this document.

4.2.2 Management Service Context

The Management Service is used by the FOS to exchange management event data with the Management Subsystem. Management events that are detected by the FOS software are sent via the Management Service and incorporated into fault, performance, security and accountability reports that MSS provides to the M&O Staff via the LSM. Network, system and operational events that are detected unintrusively by the MSS are sent via the Management Agent to the FOS. Management Event information reported by MSS is formatted by the ROS Event Handler for distribution to EOC User Station displays at the option of the user. The FOS Event Handler also makes select management event information available to FOS subsystems. FOS requirements for the MSS Management Service are provided in Section 4.2.5 of this document.

4.2.2.1 Management Events Reported to MSS

In order for fault performance security, and accountability reports to be prepared and forwarded to the LSM, management events must be detected by the ECS-developed FOS software and forwarded to the Management agent. The MSS Instrumentation service is used by FOS applications to forward application specific information that are of interest to the Management Agent. Once the instrumentation is in place FOS application software information can be gathered and metrics can be calculated automatically by the MSS service. The events collected by the Management agent are then manipulated into reports.

A representative list of faults and events reported by the various ECS managed objects at the EOC is provided in Table 4.2-1.

Table 4.2-1. Faults and Events Reported by ECS Managed Objects

Managed Object	Fault/Event
Standard SNMP Traps	Cold Startup Warm Startup Link Up Link Down Authentication Failure
Network Device	Node Added Node Deleted Node Down Node Marginal Node Unknown Node Up
Interface Card	Interface Added Interface Deleted Interface Disconnected Interface Down Interface Marginal Interface Unknown Interface Up Interface Unmanaged
Disk drive	Disk drive on-line Disk drive off-line Disk drive warning Disk drive unknown state
Printer	Printer printing Printer idle Printer warming
Tape drive	Tape drive on-line Tape drive off-line
ECS Application	Application failed Application missing Application startup Application shutdown Application discovered
ECS Database	Database up Database down

4.2.2.2 Management Events Reported to FOS

Network, system and operational events that are detected unintrusively by the MSS are sent via the Management Agent to the FOS. The FOS has two uses for this information. All of the unformatted event information received from MSS is formatted by the FOS Event Handler in for

distribution to EOC User Stations for display at the option of the user. Selected, unformatted event information is used by FOS subsystems to update ground telemetry information that supplies current statuses of hardware, software and network components to EOC User Station displays.

The EOC hardware components that are monitored on behalf of the FOS are provided in Appendix A: MSS Managed Hardware Objects.

The EOC software components monitored on behalf of the FOS include permanent as well a transient software processes. The FOS software processes that require management by this service are listed in Appendix B: MSS Managed Application (Software) Objects.

4.2.3 Management Service API

The management services provided by MSS are made available to ECS subsystems through a set of public classes. These public classes provide the interface for ECS subsystems to MSS and vice versa. These public classes, their attributes and their methods are covered in this section. They are however, described in more detail in the Release A MSS Design Specification (305-CD-013-001).

The following classes are provided as external interfaces to the management subagent for ECS application programmers.

EcAgManager

MsAgMonitor

EcAgEvent

Each of these classes are defined within this document, along with usage information and example code.

CLASS: EcAgManager

DESCRIPTION:The EcAgManager class provides mapping facilities between the ECS application and the management agent. Callback registration functions are provided by the class for the retrieval of application specific metrics (variable values). These need to be used by programmers in order to register the callbacks for this functionality. Likewise, application programmers may register a callback function for the shutdown event.

USAGE:

EcAgManager provides the application programmer one function for use in their application. They are as follows:

RegisterCallBack (int ActionType, MssCallBackFuncPtr *FuncPtr);

This function is used by the application programmer to register callback functions. The first arguments is the action in which the callback should be executed. The second function is a pointer to the actual callback function

The following are legal action types:

GET_PERFORMANCE

GET_FAULT

GET_CONFIG

SHUTDOWN

The callback function must be of the following type

typedef int (*MssCallBackFuncPtr)(void *pValue, int nIndex);

The application programmer should take note that their callback function may be called at any time after the registration of the callback and before the shutdown callback is issued. Since EcAgManager is running in a separate thread of execution, the programmer must also take any steps within the callback function to protect any critical code sections (via semaphores, file locks, ...).

Different data structures are passed in the pValue attribute depending on the action type. These structures are the following:

GET_PERFORMANCE	typedef struct { char szPerfType [17]; int nPerfValue; int nPerfThreshold } CBD_PERFORMANCE;
GET_FAULT	typedef struct { char szFaultType [17]; int nFaultValue; } CBD_FAULT;
GET_CONFIG	typedef struct { char szCfgParam [17]; char szCfgValue [17]; } CBD_CONFIG;
SHUTDOWN	int nNow;

EXAMPLE CODE:This example provides sample callbacks (shutdown, performance) and their registration with the EcAgManager class.

```
int MyShutdownCallback ( void *pValue; int nIndex )
{
 int nNow = (int)*pValue;
 if ( nNow ) {
  printf ("Shutting down immediatly\n");
  exit (0);
 }
 else {
  printf ("Shutting down later\n");
```

```
    bShutdownLater = TRUE;
  }
  return 0;
}
int MyPerformanceGetCallback ( void *pValue; int nIndex )
{
  CBD_PERFORMANCE* pPerfData = (CBD_PERFORMANCE*) pValue;
  int nReturnCode = 0;
  // critical code section, so use semaphore locking
  SEMAPHORE::Lock ( MGMTLOCK);
  switch ( nIndex ) {
    case NUMBER_WIDGETS_PROCESSED_INDEX:
      pValue->nValue = nNumberOfWidgetsProcessed;
      break;
    case NUMBER_WIDGETS_CACHED:
      pValue->nValue = nNumberOfWidgetsCached;
      break;
    default:
      nReturnCode = 1;  // no more performance attributes, so return 1
  }
  SEMAPHORE::UnLock ( MGMTLOCK );
  return nReturnCode;
}
int main ( void )
{
  EcAgManager MsManager;
  MsManager.RegisterCallback ( GET_PERFORMANCE, MyPerformanceGetCallback );
  MsManager.RegisterCallback ( SHUTDOWN, MyShutdownCallback );
  // normal application code goes here
  return 0;
}
```

CLASS: MsAgMonitor

DESCRIPTION:The MsAgMonitor class provides application programmers the ability to start and stop the management agents' monitoring of a specified process (transient processes included). Both DCE and non-DCE based application may be monitored in this fashion. Between the StartMonitor request and the StopMonitor request, in the case of a fault, a notification is sent to the

specified recipient.

USAGE:

MsAgMonitor provides the application programmer two functions for use in their application. They are as follows:

StartMonitor (int nPID, char *szDestination, char *arg3);

StartMonitor (char *arg1, char *szUUID, char *arg3);

The function start monitor has been overloaded to account for both DCE and non-DCE applications. The first instance of the method will instruct the management agent to monitor an application with the provided process ID. The second instance of the method instructs the management agent to monitor a DCE application with the provided UUID.

EXAMPLE CODE:

```
int MyClass::MySpawningOfAProcess ( )
{
  int nPID;
  // start a child application to produce widgets
  nPID = SpawnWidgetCreator ( );
  // instruct the management agent to monitor the child process
  MsMonitor.StartMonitor ( nPID, NULL, NULL );
}
```

CLASS: EcAgEvent

DESCRIPTION:The EcAgEvent class provides application programmers the ability to send application events to the management subagent.

USAGE:

EcAgEvent provides the application programmer one functions for use in their application. It is as follows:

LogEvent (char *szParentID, char *szMyId, int nEventType, int nSeverity, int nDisposition, char *szMessage)

The application programmer calls this function each time a reportable event has occurred. nEventType specifies the event type. nSeverity specifies the severity of the even. nDisposition specifies the disposition of the event. szMessage provides a brief description of the event.

EXAMPLE CODE:

```
MyClass::MyErrorHasOccured ( )
{
  // an error has occured, so report it to the management agent
  myId = EcEvent.GetMyId;
  EcEvent.LogEvent ( parentId, myId,  FAULT, WARNING, NULL,
                "cannot find widget initialization file" );
```

4.2.4 Management Service Dynamic Model

4.2.4.1 Get MIB Value Scenario

Abstract

- This scenario will demonstrate how an MSS management application checks the condition of a managed resource. That information can be obtained from a variable defined in ECS application MIB.

Interfaces

- Management Framework in MSS Fault Management

Stimulus

- An MSS management application inquires the value of a MIB variable concerning an ECS application running on a remote host.

Participating Classes

- EcAgAgent
- MsAgSubAgent
- EcAgManager
- ECS application

Pre-conditions

- The SNMP agent, ECS subagent, and the ECS application are all up and running.
- The SNMP agent on the host is up and running. It listens to port 161 to wait for incoming SNMP requests.
- The ECS subagent is functioning. It has established connections with the SNMP agent on the same host and waiits for incoming requests through the agent-subagent protocol.
- The ECS application is instrumented with the MSS-provided class library which includes the server part of the EcAgManager. This application is also up and running normally. The instrumented EcAgManager object has been instantiated.

Post-conditions

- The SNMP agent is still listening to the port 161 for incoming SNMP requests.
- The ECS subagent continuously await for incoming requests from the SNMP agent.
- The ECS application is running to perform its own functions.

Scenario description

A MSS management application needs to check the condition of a managed resource. That can be reflected by the value of a variable defined in ECS application MIB.

The management application issues an SNMP request to retrieve the value of that MIB variable. The request is passed from MSS Server to the SNMP agent on a particular remote host. The SNMP

agent first does the authentication and authorization validations. If the request is allowed to access that MIB variable, then it checks the MIB registration tree to determine which agent or subagent is responsible for that MIB variable. For accessing the application MIB variables, it passes the request to ECS subagent through the agent-subagent protocol which is SNMP MUltipleXing protocol (SMUX).

The subagent is always waiting for incoming requests from the SNMP agent. When a request comes in, it determines which access method to use for retrieving the requested MIB variable and calls that access method. The retrieved value will be passed back to SNMP agent and then relayed back to the management application on the MSS Server.

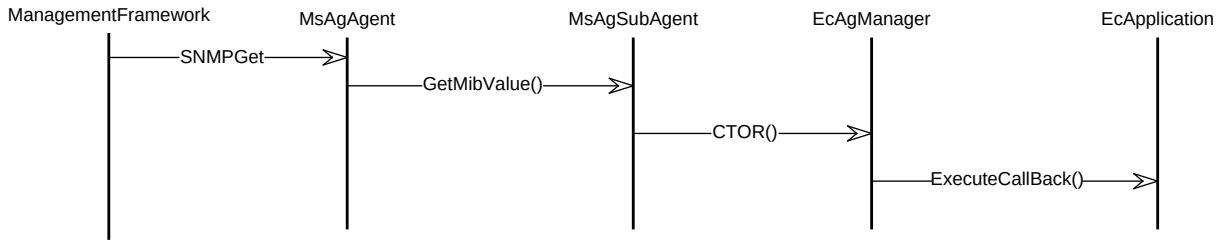

Figure 4.2.4.1-1. Get MIB Value Dynamic Model

4.2.4.2 SNMP Trap Generation Scenario

Abstract

- This scenario will demonstrate how an SNMP Trap is generated when a fault condition occurs in an ECS application.

Interfaces

- Management Framework in MSS Fault Management
- EcUtLoggerRelA in CSS

Stimulus

- An error condition occurs in an ECS application. The error is serious enough to inform the Fault Management application which runs on MSS Server.

Participating Classes

- ECS application
- EcAgAgent
- MsAgMonitor
- MsAgSubAgent
- MsAgDeputy

Pre-conditions

- ECS applications are running on a managed ECS host. The SNMP agent and ECS subagent are all up and running. The management framework is running on MSS Server.
- The ECS application is instrumented with the MSS-provided class library. When the fault condition occurs, ECS application is able to send out event notifications through the sendEvent method of EcAgEvent object.The ECS subagent is up and running on the host. It is listening to receive DCE remote procedure calls.
- The Deputy of SNMP manager (management framework) on MSS Server is up and running which is ready to receive DCE remote procedure calls.

Post-conditions

- The ECS application may or may not be running caused by the error condition.
- The ECS subagent continuous listening to receive DCE RPC calls.
- The Deputy of SNMP manager continue to listen to receive DCE RPC calls.
- An SNMP trap is generated to the management framework on MSS server. The fault management application detects the fault condition of that ECS application on that host.

Scenario description

When a fault condition occurs in an ECS application, the application instantiates an object EcAgEvent. The application invokes the sendEvent method on that object sending an event notification to MsAgSubAgent. The MsAgSubAgent will log the event to MSS log which is managed by CSS EcUtLoggerRelA. Then, the MsAgsubAgent will check the severity of the event. If it is higher than the infoLevel, then this event notification will go further to the MSS Server. The MsAgSubAgent will instantiate an object EcAgEvent and invoke its sendEvent method to send this event to the MsAgDeputy on the MSS Server. The MsAgDeputy will then in turn convert the event to an SNMP trap and send it to the management framework which is HP OpenView.

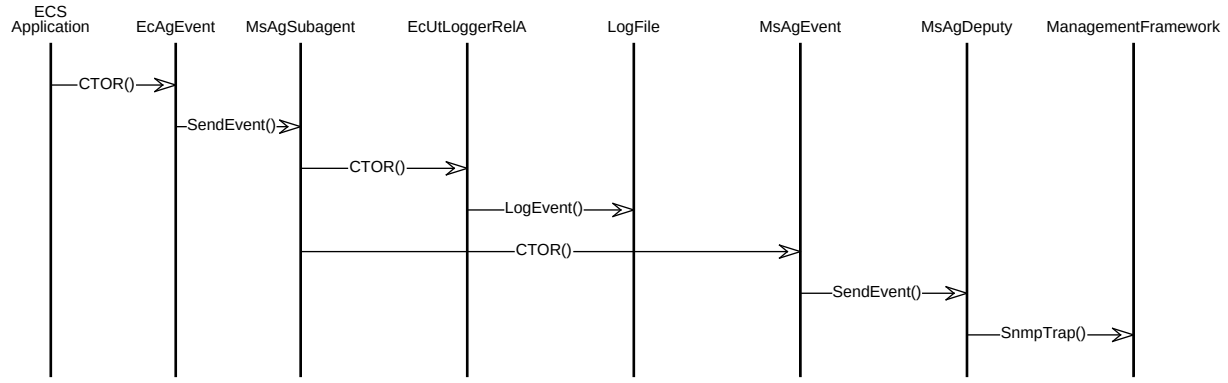

Figure 4.2.4.2-1. SNMP Trap Generation Dynamic Model

4.2.5 FOS Event API

The FOS Event service provided by the FOS Data Management Subsystem are made available to ECS subsystems through a set of public classes. These public classes provide the interface for MSS to convey management event information to FOS. These public classes, their attributes and their methods are covered in this section.

The FdEventLogger class provides appliation software a way to send events to the event handler. The user calls the GenEvent operation passing the appropriate paramenters whenever an event needs archived and sent to display. The FdEvEventLogger class will create a FoEvEvent class from the calling parameters and send the FoEvEvent class the the FdEvEventHandler class.

The FdEvEvendHandler class routes events to the FdEvEventArchiver. The FdEvEventHandler class uses the FdEvEventConfig Class to determine which events need to be sent to the FdEvEventArchiver class.

The FdEvEventConfig class contains incoming and outgoing event fileters. The user can select the type of event that need to be sent to the event archiver, and the type of events the user station needs to listen for.

The FdEvEventArchiver class receives unformatted events from event handlers. the FdEvEventArchiver uses the FdEvEventDb event databse class to determine how to format the events. The formatted events are archived using the FdEvEventFile class, and multicasted over the network to user stations. The FdEventArchiver class also uses the event databse to determine if a procedure needs initiated. If a procedure needs initiated the FdEvEventArchiver class will instantiate a FdEvProcedure class.

The FdEvEventListener class listens for formatted events on the network. The FoEvEventListener filters events by using information provided in the FdEvEventConfig class, and then sends the events to display.

4.2.6 FOS Event Dynamic Model

4.2.6.1 FOS Event Processing Scenario Abstract

The purpose of the Event Processing scenario is to describe the process by which events are generated, archived, and sent to displays. The event trace for this scenario can be found in Figure 4.2.6-1.

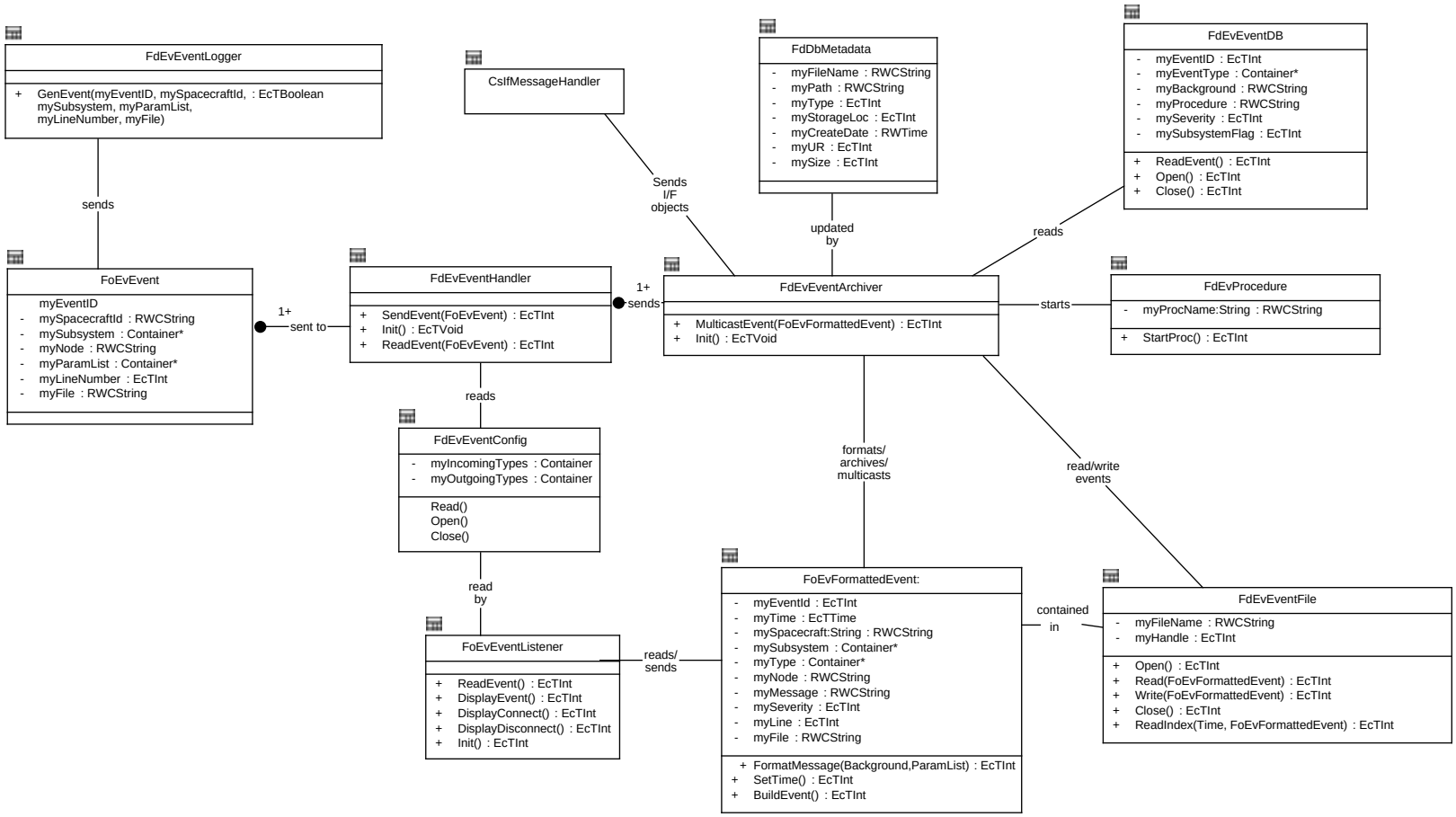

Figure 4.2.6-1. DMS Event Processing Object Model

4.2.6.2 DMS Event Processing Summary Information

Subsystem Interfaces:

CSMS:MSS

FOS Offline analysis

FOS Real-Time Telemetry

FOS Real-Time Command

FOS Real-Time Resource Management

FOS Real-Time Contact Manager

FOS Planning and Scheduling

FOS Command Management

Stimulus:

FDEventLogger genevent operation is called by application software.

Desired Response:

Formatted event is created, archived, and multicasted.

Pre-Conditions:

Event applications initialized.

Post-Conditions:

Event is stored at data server, and displayed at user station.

4.2.6.2 DMS Event processing Scenario Description

The FdEvEventLogger provides applications with a way to send events to the FdEvEventHandler. The FdEvEventLogger is responsible for creating a FoEvEvent and sending it to the FdEvEventHandler.

The FdEvEventHandler class sends the FoEvEvent class to the FdEvEventArchiver class. The FdEvEventARchiver class crates an FoEvformattedEvent by using the FdEvEvendDB event databse class, and information provided in the FoEvEvent class. The FdEvEventArchiver class will use the event id to index into the FdEvEVendDB. The FdEvEventArchiver starts the FdEvProcedure class if the FdEvEventDb class defiens a procedure to initiate. The FdEvEventArchiver archives FoEvFormattedEvent classes to the FdEvEventFile, and then multicasts FoEvformattedEvent classes over the network.

The FoEvEventListener class reads FoEvFormatted Event classes off the network. The FoEvEventLister class uses th eFdEvEventConfig class to filter events. The FoEvEventListener sends FoEvFormattedEvent classes to the event displays.

4.2.7 Management Service Requirements

1. MSS shall provide FOS with a service that shall monitor specified hardware, software and network components for changes in state.

2. MSS shall allow the FOS to register permanent and transient software processes upon creation for monitoring by the provided Management service.
3. MSS shall allow the FOS to unregister transient software processes with the Management service upon process termination.
4. MSS shall report changes in state of managed components to the FOS within five TBR seconds.
5. MSS shall report the current state of managed components to the FOS at least once every five TBR seconds whether or not the state of the managed component has changed.
6. MSS shall report management events to the FOS by providing the following information:
   a. UTC time tag
   b. Event type
   c. Event Identifier
   d. Event message
   e. Spacecraft Identifier (if applicable)
   f. Instrument Identifier (if applicable)

4.3 Performance

4.3.1 Performance Description

CSMS monitors FOS hardware and software performance. Resource availability and resource utilization data are collected on FOS resources using management agents to store the information to a local log file. Performance data that is to be compiled into management reports are sent from the log file to the management database on the Enterprise Monitoring Server via the Management Data Access service. This information can then be captured in standard or ad hoc performance reports. Examples of performance management reports include resource availability reports, resource utilization reports, and performance statistics reports. Specific performance metrics to be gathered and reported are listed in Section 4.3.5.

4.3.2 Performance Context Within FOS

CSMS uses management agents to collect FOS performance data. These agents gather performance data from the FOS applications via either operating system calls or APIs. While operating system calls are used wherever possible, since they allow for the non-intrusive monitoring of FOS applications, APIs are built into the FOS application designs to gather necessary performance data that cannot be obtained in a non-intrusive manner. All gathered performance data is logged on the FOS host, with a limited subset sent in real time to the enterprise management framework. The logged information is periodically correlated into summarized data. This summarized data is imported into the RDBMS on the Enterprise Monitoring Server for further performance analysis. The individual logs are then archived for possible future use.

4.3.3 Performance Interface Definition

The performance metrics to be collected are listed in Section 4.3.5. All of these metrics can be collected and managed by the COTS Performance Application in a non-intrusive manner.

4.3.4 Performance Dynamic Model

Applicable performance dynamic models are included in the Release A MSS Design Specification (305-CD-013-001), section 6.2.4.

4.3.5 Performance Metrics

The following performance metrics are gathered from FOS elements:

4.3.5.1 FOS Host Performance Metrics

Memory :

- amount and percentage of physical memory utilized by user
- amount and percentage of physical memory utilized by process
- amount and percentage of physical memory utilized by kernel
- amount and percentage of allocated swap space (reserve or in-use)

CPU:

- amount and percentage of CPU time utilized per process
- amount and percentage of CPU time utilized per user

Disk:

- rate of physical disk I/O's (per second)
- rate of logical disk I/O's (per second)
- rate of physical data transfer to and from disk (per second)

Network:

- rate of packets sent and received (per second)
- rate of LAN collisions (per minute)
- rate of LAN errors (per minute)

System:

- number of users logged into system
- number of processes running on system
- rate of process swaps (per minute)
- rate of system calls (per second)

Queues:

- number of processes waiting on terminal I/Os
- number of processes waiting on network activities
- number of processes in run queue
- number of processes waiting on disk I/Os
- number of processes waiting for inter-process communications, including semaphores, message queue, pipes, and sockets

4.3.5.2 FOS Process Performance Metrics

Per process:

- process ID
- CPU utilization
- memory utilization
- disk reads
- disk writes

4.4 Scheduling

The MSS Ground Events Planning Service is a complete reuse of the Production Planning Workbench that is part of the Production Planning CSCI provided by the Planning and Data Processing Subsystem (PDPS). The Production Planning Workbench is used to prepare a schedule for the production at the EOC, and forecast the start and completion times of the activities within the schedule. The functions provided by the workbench include creating a candidate plan from production requests, activating a candidate plan, updating the active plan, and canceling or modifying the active plan.

The Production Planning Workbench provides an interface to submit operations ground events such as production, maintenance, and testing. It develops optimum resource utilization plans and schedules based upon approved system configurations and priorities.

The Production Planning Workbench is used to plan ground events for all site resources, including FOS resources at the EOC. The flight operations schedules generated by the FOS planning and scheduling function are be used by the Ground Event Planner as input to resource planning for ground events support. Once the flight operations schedules have been received, the Ground Event Planner generates a ground event schedule based on resource availability. In addition the ground event schedule and allocated resources are available to EOC operations staff and are expected to be closely coordinated during the planning process. The design of the Planning Workbench supports multiple users and therefore can be used by the EOC operations staff to allocate resources to specific missions.

4.5 Configuration Management

The Configuration Management Application Service (CMAS) provides tools with which the ECS staff at the EOC tracks deployed ECS baselines and controls changes to the hardware and software that comprise them.

CMAS maintains electronic stores of baseline data, software, and system change requests that enter the operational environment, making them and a variety of reports available for system maintenance and operations activities. It accepts ECS and algorithm software and non-real time configuration management data from formatted files or via operator interface. M&O staffs, sustaining engineers, and AIT teams rely on CMAS data stores to make, track and audit configuration changes and to help enforce ECS CM rules. They also use CMAS to produce formatted files containing change requests, site baseline records, software, documentation, and reports that can be made available for distribution system-wide via CSS services such as e-mail, ftp, and the ECS bulletin board.

The CMAS includes three service managers:

Baseline Manager - The Baseline Manager is a COTS application that helps the EOC M&O staff manage ECS baseline records. The Baseline Manager COTS application is an interactive product that facilitates the tracking of baselines. ECS-customized portions of the Baseline Manager include reports, tailored input forms, interfaces to ClearCase and DDTS, and triggers for system management activities.

Software Change Manager - The Software Change Manager is a software application that helps the EOC M&O staff organize and partition software, control software changes and versions and assemble sets of software for release purposes. ClearCase, a COTS product, has been selected to perform the Software Change Manager functions. ClearCase provides extensive software library management facilities. ECS-customized portions of the Software Change Manager include reports, triggers to implement policies and provide an interface with DDTS and the Baseline Manager, scripts for frequently recurring operations, and views for providing access to related files.

Change Request Manager - A software application that enables the EOC staff to register and keep track of configuration change requests (CCR), non-conformance (NCR), and deficiency reports (DR) electronically. The Distributed Defect Tracking System (DDTS), a COTS product, has been selected to perform the Change Request Manager functions. DDTS provides interactive functionality for tracking CCRs and NCRs. ECS-customized portions of the Change Request Manager include forms and reports, tailoring of rules and event flow logic to the ECS environment, and interfaces with ClearCase and the Baseline Manager.

4.6 Other Services

Other MSS services provided to FOS are Security Management, Accountability Management, and Trouble Ticketing. These services do not interface directly with FOS applications other than the Security Management and Accountability Management event notifications which are described in the Management Services section (Section 4.2).

4.6.1 Security Management Service

The Security Management Application Service provides for the management of the security mechanisms that are used to protect and control access to ECS resources at the EOC. It provides the rules and the implementation for authentication procedures, the maintenance of authorization facilities, the maintenance of security logs, intrusion detection and recovery procedures. The mechanisms used to provide security in ECS comprise three distinct parts: network security, distributed communications security, and host-based security.

Network security management involves the management of routing tables used for address-based filtering (network authorization). This is implemented through router COTS configuration files through which access control rules are specified.

Distributed communications security addresses communications between software entities such as clients and servers employing mechanisms such as Kerberos/DCE for real-time authentication exchange. The management of distributed communications security involves the management of the authentication database (the DCE registry database), the authorization database (DCE Access Control List Managers). This is implemented through the use of HAL DCE Cell Manager. The DCE Cell Manager is a COTS product that provides a Motif-based capability to administer the DCE security registry (authentication database), and the access controls on cell resources (authorization database).

Host-based security management addresses the control of access to and the protection of these mechanisms, in addition to the management of compliance to established security policy (e.g.,

password usage guidelines), and intrusion detection (e.g. break-ins). Access control to network services is implemented through TCP wrappers, a public domain tool. Compliance management is implemented through public domain products npasswd, crack, and SATAN. Intrusion detection is implemented through the public domain product Tripwire, and custom development.

The EOC Security Management Application Service manages local security databases, manages compliance to security directives and guidelines established and disseminated by the SMC, performs intrusion detection checks in order to maintain the integrity of ECS resources, provides the capability to analyze security audit trails, and provides the mechanisms to generate reports for such these activities. The SMC Security Management Application Service is responsible for establishing and disseminating security guidelines to the sites, disseminating security advisories received from external systems (security agencies such as CERT and NIST) to the sites, receive security reports from the sites, and to receive notifications of and coordinate the recovery from detected security breaches at the sites and external systems.

Notifications of security events and summary data are forwarded by the Security Management Application Service to the SMC, while coordination for recovery and security advisories are received from the SMC. An interface to the Fault Management Application Service (via the Management Agent Service) allows for the Security Management Application Service to send security event notifications, fault events, and receive startup and shutdown commands. The use of event notifications by the Security Management Application Service is described in the Management Services section (Section 4.2).

4.6.2 Accountability Management Service

The Accountability Management Service provides the capabilities of User Registration and the generation of reports from audit trails.

The EOC provides for the registration of the IST class of users. The registration service provides the capabilities for the creation, modification and maintenance of accounts with user profiles.

The user profile contains information about the user. This includes the name of the user, a user identification code, the user's primary DAAC, the organizational affiliation, investigating group (such as an instrument team) affiliation (if any), the project the user is affiliated with, the name of the PI of the project, the mailing address of the user, the shipping address to which data needs to be sent, media preferences for orders, the user's telephone number and the user's electronic mail address (if any).

The Accountability Management Service makes the user profile available , such as the Data Server, information such as the user's electronic mail address and the shipping address, which are used for the distribution of data.

The Audit Trail capability provides the means to verify the integrity of the system. This comprises the generation of a user audit trail and a security audit trail with data collected from a variety of sources. The data used to generate these audit trails is captured to local log files via the event logging services described in the Management Services section (Section 4.2).

4.6.3 Trouble Ticketing Service

The Trouble Ticketing Service (TTS) provides the EOC as well as the DAACs with a common environment and means of classifying, tracking, and reporting problem occurrence and resolution

to both ECS users and support staff members. TTS's core functionality is provided by the Remedy Action Request System, a COTS product. Through the configuration of this product, TTS:

- provides a graphical user interface for support staff members to access all TTS services
- includes a definition of the common trouble ticket entry format
- stores trouble tickets
- retrieves trouble tickets through a wide variety of criteria (ad-hoc queries)
- provides the ability to "forward" problems from the EOC to another ECS location (DAAC or SMC)
- produces stock and common reports
- interfaces with the common e-mail environment to provide automatic notification to users and support staff members
- offers an application programming interface through which applications could submit trouble tickets
- provides summary information to the SMC to allow trend reports regarding trouble tickets.
- defines a consistent "life-cycle" for trouble tickets (through a set of standard status codes and escalation and action rule definition)
- allows the EOC (and each of the other sites) a degree a customization through definition of further escalation and action rules

Escalation rules are simply time activated events which execute on trouble tickets which meet a set of specified criteria. Actions which can be taken include notification (of either a user or support staff member), writing to a log file, setting a field value on the trouble ticket, or even running a custom written process. Qualifications can be expressed on any trouble ticket data TTS tracks. Examples of custom escalation rules might include...

> if a "High" priority trouble ticket stays in "Assigned" for more than 48 hours without being moved to "In-Progress", re-notify the assigned support staff member
>
> if a "Low" priority trouble ticket is not moved to "Closed" within 14 days, raise the priority to "Medium" and re-notify the assigned support staff member

Active links are similar to escalation rules with the exception that they are defined to take place on a specified action rather than at a given time. Examples of custom active links which can be defined by the EOC include...

> if a high priority trouble ticket is closed with a particular resolution code, notify a specified member of the support staff (perhaps a manager)

In addition to the functionality provided by Remedy, TTS utilizes a set of custom HTML documents to provide users with the ability to submit new trouble tickets and query the current status of any of their previous entries. Access to TTS through this technique provides users an easy method for reporting problems in an environment with which most are already familiar. Additionally, as another means of trouble ticket entry, the TTS provides a textual e-mail template through which automated entry of trouble tickets is also possible. Finally, support staff members are able to enter trouble tickets through the Remedy provided interface for problem received via other methods (e.g. phone calls).

This page intentionally left blank.

5. SDPS Services

5.1 FOS/SDPS Interface Overview

5.1.1 Overview

This section of the ICD provides a FOS-specific description of the data services provided by the SDPS to the FOS. Specific information (e.g., type, frequency, volume) for data exchanges is presented.

5.1.2 SDPS/FOS Interface Description

The SDPS supports the ingest of data into ECS storage repositories. The SDPS supports a diverse set of internal and external interfaces. A variety of data structures and formats must be supported, each with potentially unique aspects. Therefore, the SDPS provides ingest clients to handle each internal and external interface. In particular, an ingest client will be established to support the interface with the FOS Data Management Subsystem (DMS) for the ingest of FOS data files. The SDPS Data Server provides long-term archiving services for the FOS. The data to be archived includes the spacecraft and instrument real-time housekeeping telemetry files, ground telemetry (CODA and NCC monitor blocks), event files, configuration data files, operational data files, and plans and schedules.

Once data are stored in the SDPS, the FOS may request retrieval of the data. The SDPS provides the set of public services to allow data retrieval.

The FOS DMS provides services for database update and retrieval, file management, and data archival and retrieval. The FOS Data Management (FDM) archive maintains the telemetry data, event data, and operational data files for a nominal period of one day. After one day the data is sent to SDPS for permanent archive. The data is maintained in the FDM archive for a minimum of 7 days, but the data may be purged from the FDM archive after 7 days if confirmation of successful storage by the Data Server is received. A summary of the routine data types, frequency of transfer, and file sizes to be transferred between the FOS EOS Operations Center (EOC) and the SDPS Ingest Clients is contained in Table 5.1-1. The types of data transferred between SDPS and FOS are shown in Table 5.1-1.

Table 5.1-1. FOS Interface Data Type Description (AM-1) (1 of 2)

Source	Destination	File type	# of files	Freq./ day	File size (MB)	Comments
SDPS	EOC DMS	Real-time Telemetry	4	6	7.2	Merged Housekeeping
SDPS	EOC DMS	Ground Telemetry	2	24	1	CODA, NCC Monitor Blocks
SDPS	EOC DMS	Events	1	24	1	Event file every hour

Table 5.1-1. FOS Interface Data Type Description (AM-1) (2 of 2)

Source	Destination	File type	# of files	Freq./ day	File size (MB)	Comments
SDPS	EOC DMS	Configuration Data	30	weekly	5	Databases, Documents, Templates, Definitions, Network Config
SDPS	EOC DMS	Operational Data	2	20	1	Statistics, Reports, Logs, Request Files, Snaps, Analysis Results, Procedures, Dumps
SDPS	EOC DMS	Plans&Schedules	10	1	5	Schedules, Profiles,LTSP,LTIP, FDF Data
EOC DMS	SDPS	Real-time Telemetry	4	6	7.2	Merged Housekeeping
EOC DMS	SDPS	Ground Telemetry	1	1	1	CODA, NCC Monitor Blocks
EOC DMS	SDPS	Events	1	24	1	2 hours of event data 5 times a day
EOC DMS	SDPS	Configuration Data	5	monthly	1	Databases, Documents, Templates, Definitions, Network Config
EOC DMS	SDPS	Operational Data	1	50	1	Statistics, Reports, Logs, Request Files, Snaps, Analysis Results, Procedures, Dumps
EOC DMS	SDPS	Plans&Schedules	2	1	5	Schedules, Profiles,LTSP,LTIP, FDF Data
SDPS	EOC DMS	Engineering				Engineering data for each instrument (from the Level 0 data)

5.2 Data Exchange Framework

Section 5.2 defines the data exchange framework for the network interface, message flows, and file transfers between SDPS and FOS. Section 5.2.1 describes the network interface, including routers and IP addresses, file transfer mechanism, and security considerations. Section 5.2.2 describes the control messages exchanged between the two systems for the necessary handshaking, and the procedure for file transfers.

5.2.1 SDPS-FOS Network Interface

5.2.1.1 Network Protocol

The network protocol used for FOS/SDPS communication is the Internet Protocol (IP), specified in RFC 791, which routes the data via the GSFC exchange LAN. The topology of the network interface between FOS and SDPS is **TBD** due to the EBnet consolidation. The IP addresses and routers for the FOS and SDPS nodes in the network are **TBD**.

5.2.1.2 Transport Protocol

The transport protocol used for FOS/SDPS communication is the Transmission Control Protocol (TCP), providing reliable delivery of data. TCP transports data between the network and applications. TCP is specified in RFC 793.

5.2.1.3 File Transfer

All file transfers are conducted using the kerberos file transfer protocol (kftp) "get" or "mget" commands, "get" for transferring one file at a time and "mget" for transferring many files at once. Kerberos is described in RFC 1510. All data exchanges between SDPS and FOS are to be automated and pulled by the consumer system (Computer Based Interface (CBI) Get).

5.2.1.4 Security Considerations

File transfer security is handled by the kerberos ftp, and is documented in RFC 1510.

To maintain security integrity, UserIDs and/or Passwords are changed periodically (at least every 6 months).

SDPS will enforce this through Unix-like group and user permissions, described in Section 5.2.2.2 of the document 305-CD-012-001, "Release A CSMS Segment Communications Subsystem Design Specification for the SDPS Project".

FOS has the authority to order data, to get the status of these Data Requests, and to cancel Data Requests.

5.2.2 Handshaking Control Messages and File Transfer Sequences

Initiation and completion of data transfer requires transmission of control messages between SDPS and FOS. Handshaking control messages between FOS and SDPS are employed by initiating an application program-to-application program TCP/IP connection. The initiator of the connection is the controller of the session, and is also the terminator of the connection. Table 5.2-1 lists the messages involved before and after an electronic "mget" transfer. The specific messages which comprise the handshaking procedure are defined in Sections 5.2.3.1 through 5.2.3.12.

Table 5.2-1. Control Messages (1 of 2)

Message Name	Message Purpose	Description
Authentication Request	Session Establishment	System establishing TCP/IP connection requests authentication; required immediately after session establishment
Authentication Response	Session Establishment	Response to Authentication Request, containing Authentication Check results
Data Availability Notice (DAN)	Notification of Data Ready for Transfer	System with data notifies Consumer system that the data are staged and ready for transfer.
Data Availability Acknowledgment (DAA)	DAN Handshake	Consumer system acknowledges that the DAN has been received, and notifies of any DAN errors

Table 5.2-1. Control Messages (2 of 2)

Message Name	Message Purpose	Description
Data Delivery Notice (DDN)	Notification of Data Transfer	Consumer system notifies that data has been transferred, ingested and archived; includes identification of data retrieval success and/or problems
Data Delivery Acknowledgment (DDA)	DDA Handshake	System with data notifies Consumer system that the DDN has been received, which indicates that the data can be deleted
Data Request	Request data	FOS requests data from the SDPS archive
Data Request Acknowledgment	Data Request Handshake	SDPS acknowledges to FOS that the Data Request has been received and is or is not valid
Data Request Status Request	Request status	FOS requests the status of a Data Request
Data Request Status	Status	SDPS provides FOS with the status of a Data Request
Data Request Cancellation Request	Cancellation	FOS requests cancellation of a Data Request
Data Request Cancellation	Cancellation Notification	SDPS notifies FOS that the Data Request has been canceled.

5.2.2.1 SDPS Retrieves Data From FOS to Archive

The transfer session is established through the exchange of an Authentication Request and Response. Authentication is done immediately upon establishment of a connection, by sending an Authentication Request message and receiving an Authentication Response message.

After successful authentication, FOS sends a Data Availability Notice (DAN) message to SDPS, specifying the names of the data files, file sizes, file dates and times, number of files, and file locations for the files available for SDPS to archive. SDPS sends the corresponding handshake/ control message, the Data Availability Acknowledgment (DAA), which reports the disposition of the DAN. SDPS then schedules to pull the data.

At the scheduled time (usually immediately), SDPS begins the "(m)get" kftp file transfer process, and transfers all the files listed in the DAN. Each file name and size is checked against DAN information, and the file transfer result is logged for the Data Delivery Notice (DDN). After all the files have been transferred, ingested, and archived or attempts exhausted, SDPS sends FOS a DDN notifying whether all files were successfully archived or the reason for failure of the "mget". FOS responds back with the corresponding handshake/control message, the Data Delivery Acknowledgment (DDA). In the case of failure in file transfer and archive, FOS sends a new DAN when the errors have been corrected.

It is possible to send more than one DAN within a TCP/IP session; however they cannot be sent until the acknowledgment (DAA) is received for the previous DAN. Each DAN is distinguished from the others by the sequence number and processor identifier which created it.

In cases of problems in file transfers (kftp), data transfer attempts are repeated an operations tunable number of times. However, if the problem cannot be resolved within a mutually determined time frame, SDPS and FOS operations personnel have the option to coordinate data delivery on 8 mm tapes.

5.2.2.2 FOS Requests/Retrieves Data From SDPS

The transfer session is established through the exchange of an Authentication Request and Response. Authentication is done immediately upon establishment of a connection, by sending an Authentication Request message and receiving an Authentication Response message.

After successful authentication, FOS sends a Data Request message to SDPS, ordering the data needed for reprocessing. SDPS sends the corresponding handshake/control message, the Data Request Acknowledgment (DRA), which reports the disposition of the Data Request. SDPS then retrieves the requested data, if available, as soon as possible from the archive and places it on the file server.

After successful authentication, SDPS sends a DAN message to FOS, specifying the number of files, file sizes, file names and directory paths for the files available for FOS to kftp from the SDPS file server. If the number of files is less than the number requested in the Data Request, then some of the requested files were not found in the archive. There is one DAN sent for each Data Request. FOS sends the corresponding handshake/control message, the Data Availability Acknowledgment (DAA), which reports the disposition of the DAN. FOS then schedules to pull the data.

At the scheduled time (usually immediately), FOS begins the "(m)get" kftp file transfer process, and transfers all the files listed in the DAN. Each file name and size is checked against DAN information, and the file transfer result is logged for the Data Delivery Notice (DDN). After all the files have been transferred successfully (but not verified) or transfer attempts have been exhausted, FOS sends SDPS a DDN notifying whether all files were successfully transferred or the reason for any transfer failure. SDPS responds back with the corresponding handshake/control message, the Data Delivery Acknowledgment (DDA).

It is possible to send more than one DAN within a TCP/IP session; however they cannot be sent until the acknowledgment (DAA) is received for the previous DAN. Each DAN is distinguished from the others by the sequence number and processor identifier which created it.

In cases of problems in file transfers (kftp), data transfer attempts are repeated an operations tunable number of times. However, if the problem cannot be resolved within a mutually determined time frame, SDPS and FOS operations personnel have the option to coordinate data delivery on 8 mm tapes.

5.2.2.3 FOS Requests Status of Data Request From SDPS

When FOS needs status of a Data Request, it sends a Data Request Status Request to SDPS. SDPS then sends a Data Request Status message.

5.2.2.4 FOS Cancels Data Request

When FOS needs to cancel a Data Request, it sends a Data Request Request message to SDPS. SDPS then sends a Data Request Cancellation message.

5.2.3 Message Format and Contents Overview

The control messages identified in Table 5.3-1 vary in purpose, length and format. Some are strictly handshaking messages, while others relate needed information to support data transfer. The message formats contain both fixed and variable length strings. A zero byte (NULL character) is used as a field separator in the manner of the C programming language. Field lengths are specified in terms of bytes, where a byte is equal to 8 bits. The specified field lengths do not include the null character used to terminate variable length strings.

The order of transmission of a group of bytes is the normal order in which they are read in English. Whenever a byte represents a numeric quantity, the left most bit in the diagram is the high order or most significant bit. Similarly, whenever a multi-byte field represents a numeric quantity, the left most bit of the whole field is the most significant bit; the most significant byte is transmitted first.

A control message is rejected when it contains errors or is sent in an inappropriate sequence. The message source receives notification of this rejection from the message destination. Error conditions for each of the messages include out-of-bound parameter values, invalid parameter values, and missing parameter values (e.g., message type). In most cases, the message is corrected by the message source, and resent. Rejection of an Authentication Request also causes the TCP connection to be broken.

The message transfer scenario between FOS and SDPS supports operator tunable parameters. Operator tunable parameters include:

- Number of authentication attempts
- Time between authentication attempts
- Time either system waits for a DAN's corresponding DAA prior to resending the DAN
- Time either system waits for a DDN's corresponding DDA prior to resending the DDN
- Time FOS waits for a Data Request Acknowledgment, Data Request Status, and Data Request Cancellation prior to resending the Data Request, Data Request Status Request, and Data Request Cancellation Request.
- Time between SDPS sending DAN to FOS and the time the data will be deleted from the SDPS file server.

5.2.3.1 Authentication Request

Table 5.2-2 shows the contents and format of an Authentication Request. An Authentication Request is a request from the Originating System to be verified as a valid user of the Destination System. An Authentication Request is the first message sent by the originator of the TCP session prior to transmission of any other data transfer message. It is used during the Authentication Check to validate the TCP session by verifying the requester's access and is performed immediately after the establishment of each TCP session. If the Authentication Request is rejected, the connection is broken. An authentication request fails if any of the message fields are invalid.

Table 5.2-2. Authentication Request Message Definition

Field	Description	Type (Length in Bytes)	Value
Message Type	Authentication Request	Unsigned Integer (1 B)	15
Message Length	Length of Message (L) in Bytes (non-zero integer)	Unsigned Integer (3 B)	≤ 84
Destination System ID	Communications Server	ASCII String (≤ 20 B)	processor id
Origination System ID	Communications Client	ASCII String (≤ 20 B)	processor id
UserID	User-provided identification; assigned by Destination system	ASCII(≤ 20 B)	UserID
Password	Authentication parameter - password assigned to User by destination system	ASCII (≤ 20 B)	Password

5.2.3.2 Authentication Response

After the Destination System performs an Authentication Check on the Origination System, it returns an Authentication Response. This response is used to relate the results of the TCP/IP session validation process. If a service was invoked before authentication was performed, then an Authentication Response message is returned with a disposition value of 2, indicating rejection. Other conditions causing rejection of an Authentication Request are discussed in Section 5.2.3. Table 5.2-3 defines the format and contents of the Authentication Response message.

Table 5.2-3. Authentication Response Message Definition

Field	Description	Type (Length in Bytes)	Value
Message Type	Authentication Response	Unsigned Integer (1 B)	16
Message Length	Length of Message in Bytes (non-zero integer)	Unsigned Integer (3 B)	≤ 45
Destination System ID	Communications Server	ASCII String (≤ 20 B)	Same as Origination System in Authentication Request
Origination System ID	Communications Client	ASCII String (≤ 20 B)	Same as Destination System in Authentication Request
Authentication Disposition	Result of authentication	Integer (1 B)	1 - accepted 2 - rejected

5.2.3.3 Data Availability Notice (DAN)

A DAN message is sent by the system supplying the data to the Consumer System to announce the availability of data for transfer. It specifies the parameters needed to identify what files are ready for pickup, their location, and how long they will be available in that location. The maximum message length allowed for a DAN is 1 megabyte (1,048,576 bytes).

Each DAN includes a Message Header, Exchange Data Unit (EDU) Label, a DAN Label and Parameter Value Language (PVL) Statements. Table 5.3-4 contains the Message Header and

Standard Formatted Data Unit (SFDU) labels; Table 5.3-5 specifies the required parameters in the DAN PVL and their values, for DANs from SDPS to FOS (where the Consumer System is FOS) and from FOS to SDPS (where the Consumer System is SDPS).The DAN PVL statements are ASCII strings, having at most 256 characters, in the form of: "Parameter = Value;". The value strings shown in Table 5.3-5 include pre-defined values enclosed within single quote marks and processor determined values. Processor determined values include ASCII alphanumerics, ASCII numerics, and International Standards Organization (ISO) times to be filled in with the proper values by the originating system's processor during DAN creation. The combination of the DAN sequence number and originating system (processor identifier) parameters uniquely identify each DAN and provides the link between related DAN, DAA, DDN, DDA, and Data Request control messages. The FILE_SPEC and FILE_GROUP objects are repeatable within a single DAN, for multiple files and/or file groups. The TOTAL_FILE_COUNT parameter indicates the number of files staged for retrieval; if this number is less than the number of files ordered in a Data Request it indicates that the some of the files were not found in the archive.

Table 5.2-4. DAN Message Header, and EDU and DAN Labels

Field	Description	Type (Length in Bytes)	Value
Message Header (4 Bytes)			
Message Type	indicates DAN	Unsigned Integer (1 B)	1
Message Length	Length of Message in bytes	Unsigned Integer (3 B)	≤ 1,048,576
Exchange Data Unit (EDU) Label (20 Bytes)			
Control Authority ID	Not used	ASCII (4 B)	'0000'
Version ID	Not used	ASCII (1 B)	'0'
Class ID	Class of label	ASCII (1 B)	'Z'
S1	Not used	ASCII (1 B)	'0'
S2	Not used	ASCII (1 B)	'0'
Data Description	EDU Indicator	ASCII (4 B)	'0001'
Delimitation Parameter	Length in ASCII of DAN Label and PVL statements, including white space	ASCII (8 B)	≤ 1,048,576
DAN Label (20 Bytes)			
Control Authority ID	Not used	ASCII (4 B)	'0000'
Version ID	Not used	ASCII (1 B)	'0'
Class ID	Not used	ASCII (1 B)	'0'
S1	Not used	ASCII (1 B)	'0'
S2	Not used	ASCII (1 B)	'0'
Data Description	Not used	ASCII (4 B)	'0000'
Delimitation Parameter	Length in ASCII of PVL statements, including white space	ASCII (8 B)	≤ 1,048,576

Table 5.2-5. Required DAN PVL Parameters (1 of 2)

Parameter	Description	Type(Length in Bytes)	Consumer System	Value
ORIGINATING_ SYSTEM	Originator of DAN	ASCII (20 B)	FOS or SDPS	FOS Processor Identifier (see Note)
CONSUMER_ SYSTEM	Destination of DAN	ASCII (20 B)	FOS or SDPS	SDPS Processor Identifier (see Note)
DAN_SEQ_NO	Sequence number assigned by originating system	ASCII (10 B)	FOS or SDPS	≤ 9999999999
REQUEST_ID	Identifier of corresponding Data Request	ASCII (20 B)	FOS	REQUEST_ID in Data Request Acknowledgment
PRODUCT_ NAME	Name of Product which defines the collection of files comprising of the product.	ASCII (25 B)	FOS	SDPS Product ID
MISSION	Mission or investigation which includes the sensors producing the data of this notice	ASCII (20 B)	FOS	'FOS', 'DMSP', etc.
REQUEST_ TYPE	Type of request which applies to this DAN	ASCII (12 B)	FOS	'Data Request'
TOTAL_FILE_ COUNT	Total number of files to transfer	ASCII (4 B)	FOS or SDPS	0 - 9999 (0 means that no files were found matching Request)
AGGREGATE_ LENGTH	Total number of bytes to transfer (sum for all files)	ASCII (10 B)	FOS or SDPS	≤ 9999999999
EXPIRATION_ TIME	ISO Time for data deletion from originating system	ASCII (20 B)	FOS or SDPS	yyyy-mm-ddThh:mm:ssZ, where T and Z are literals (TBD hours after DAN sent)
OBJECT	Start of file group parameters (repeat for each group of files)	ASCII (10 B)	FOS or SDPS	'FILE_GROUP'
DATA_TYPE	SDPS Data Type	ASCII (20 B)	FOS or SDPS	'LEVEL1', 'LEVEL2', 'LEVEL3'
DESCRIPTOR	Name of sensor or instrument that collected the data	ASCII (4 B)	FOS or SDPS	'TMI', 'VIRS', 'PR', 'GV'
DATA_VERSIO N	Version of Data files	ASCII (4 B)	FOS or SDPS	'V001' through 'V999',
OBJECT	Start of file parameters (repeat for each file)	ASCII (9 B)	FOS or SDPS	'FILE_SPEC'
NODE_NAME	Name of network node on which the file resides	ASCII (30 B)	FOS or SDPS	e.g.. 'shark.hitc.com'

Table 5.2-5. Required DAN PVL Parameters (2 of 2)

Parameter	Description	Type(Length in Bytes)	Consumer System	Value
DIRECTORY_ID	File directory name (i.e., path name)	ASCII (256 B including FILE_ID, but excluding null terminator)	FOS or SDPS	e.g./PR/Level1/
FILE_ID	File name	ASCII (256 B including DIRECTORY_ID, but excluding null terminator)	FOS or SDPS	file name
FILE_TYPE	File Data Type	ASCII (20 B)	FOS or SDPS	e.g.,'BROWSE', 'IMAGE', 'METADATA', 'CALIBRATION'
FILE_SIZE	Length of file in bytes	ASCII (10 B)	FOS or SDPS	≤ 9999999999
BEGINNING_DATE/TIME	ISO Start time of data in file as defined in the metadata	ASCII (20 B)	FOS or SDPS	yyyy-mm-ddThh:mm:ssZ where T and Z are literals
ENDING_DATE/TIME	ISO End time of data in file as defined in the metadata	ASCII (20 B)	FOS or SDPS	yyyy-mm-ddThh:mm:ssZ, where T and Z are literals
END_OBJECT	End of file parameters (repeat for each file)	ASCII (9 B)	FOS or SDPS	'FILE_SPEC'
END_OBJECT	End of file group (repeat for each file group)	ASCII (9 B)	FOS or SDPS	'FILE_GROUP'

Note. Each processor must have a unique identifier.

5.2.3.4 Data Availability Acknowledgment (DAA)

A DAA message is the corresponding handshake/control message for the DAN. The DAA acknowledges receipt of the DAN and provides the mechanism to identify the status of data transfer scheduling and any DAN errors. The short form of the DAA is used for both error-free DANs and DANs with header and label errors. A long form of the DAA message is used when some file groups in the DAN have invalid parameters. Other conditions causing rejection of a DAN are discussed in Section 5.3.3. The format and content of the short and long DAA messages is defined in Tables 5.3-6 and 5.3-7, respectively.

Table 5.2-6. Short DAA Message Definition

Field	Description	Type (Length in Bytes)	Value
Message Type	Short Data Availability Acknowledgment	Unsigned Integer (1 B)	2
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	19
DAN Sequence No.	Sequence number assigned by DAN sender	ASCII (10 B)	DAN_SEQ_NO in DAN
Disposition	Disposition Bits	Logical Bits (4 B)	all 0 - accepted bit 0 - spare bit 1-invalid DAN sequence number bit 2-spare bit 3-invalid mission ID bit 5.3- spare bit 5-invalid file count bit 6-invalid data service bit 7-other errors bit 8 - EDU label error bit 9 - DAN label error bit 10 - invalid DAN length bit 11 - invalid aggregate length
Transfer Start Time	Not used	Integer (1 B)	Null

Table 5.2-7. Long DAA Message Definition (1 of 2)

Field	Description	Type (Length in Bytes)	Value
Message Type	Long Data Availability Acknowledgment	Unsigned Integer (1 B)	3
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	≤ 1,048,576
DAN Sequence No.	Sequence number assigned by DAN sender	ASCII (10 B)	DAN_SEQ_NO in DAN
Number of File Groups (to follow)	Number of File Groups with Errors	ASCII (4 B)	Number of File groups, in DAN, with errors

Table 5.2-7. Long DAA Message Definition (2 of 2)

Field	Description	Type (Length in Bytes)	Value
For each file group having errors			
Data Type PVL	SDPS Data Type	ASCII String (≤ 20 B)	DATA_TYPE in DAN
Descriptor PVL	Name of instrument/ sensor that collected the data	ASCII String (≤ 60 B)	DESCRIPTOR in DAN
Disposition	Disposition bits	Logical Bits (2 B)	bit 0 - not used bit 1 - invalid data type bit 2 - not used bit 3 - invalid descriptor bit 4 - invalid directory bit 5 - not used bit 6 - not used bit 7 - not used bit 8 - invalid file size field bit 9 - invalid file ID bit 10 - invalid time/data format bit 11 - invalid version # bit 12 - invalid node name

5.2.3.5 Data Delivery Notice (DDN)

A DDN is sent from the system which has completed retrieving the files via kftp from the supplier of the data. The DDN announces the completion of data transfer and archival, and identifies the success of file transfers and any errors or problems that occurred. The short DDN is used for notification of error-free data transfers and mainly communication related errors. If all files in a request do not have the same disposition, a long form of this message is employed. The format and content of the short and long DDN messages is defined in Tables 5.2-8 and 5.2-9, respectively.

Table 5.2-8. Short DDN Message Definition

Field	Description	Type (Length in Bytes)	Value
Message Type	Short Data Delivery Notice	Unsigned Integer (1 B)	11
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	46
DRR Sequence No.	Not used	Integer (4 B)	0
DAN Sequence No.	Sequence number assigned by DAN sender	ASCII (10 B)	DAN_SEQ_NO parameter in DAN
Disposition	Successful Network Failure Unable to Establish FTP Connection Host Denied Access All File Groups/Files not found FTP failure - Too many errors in file transfer Post-transfer double-check failed FTP command error	Integer (1 B)	0 1 2 3 4 5 6 7
Spares		(3 B)	
Time Stamp	ISO Time when Consumer System transferred the last part of the data	ASCII (20 B)	GMT in the following format: yyyy-mm-ddThh:mm:ssZ, where T and Z are literals
Throughput	Rate in bytes per second averaged over all files	Integer (4 B)	≥0 (0 indicates unsuccessful transfer)

Table 5.2-9. Long DDN Message Definitions (1 of 2)

Field	Description	Type (Length in Bytes)	Value
Message Type	Long Data Delivery Notice	Unsigned Integer (1 B)	12
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	≤ 1,048,576
DRR Sequence No.	Not used	Integer (4 B)	0
DAN Sequence No.	Sequence number assigned by DAN sender	ASCII (10 B)	DAN_SEQ_NO parameter in DAN
Number of Files	Number of Files in DAN	ASCII (4 B)	TOTAL_FILE_COUNT parameter in DAN
For each File			
File Directory	ASCII string specifying file directory name (i.e. path name)	ASCII (≤ 256 B) (including FILE_ID but excluding the null terminator)	DIRECTORY_ID parameter in DAN
File Name	File names on system sending DAN	ASCII (≤ 256 B) (including DIRECTORY_ID but excluding the null terminator)	FILE_ID parameter in DAN

Table 5.2-9. Long DDN Message Definitions (2 of 2)

Field	Description	Type (Length in Bytes)	Value
File Transfer Disposition	One of the following:	Integer (1 B)	
	Successful		0
	Network Failure		1
	Unable to Establish FTP Connection		2
	Host Denied Access		3
	File not found		4
	FTP failure - Too many errors in file transfer		5
	Post-transfer double-check failed		6
	All other failure conditions		7
Time Stamp	ISO Time when Consumer System transferred the last part of the data	ASCII (20 B)	GMT in format yyyy-mm-ddThh:mm:ssZ, where T and Z are literals
Throughput	Rate in bytes per second for each file	Integer (4 B)	≥ 0 (0 indicates unsuccessful transfer)

5.2.3.6 Data Delivery Acknowledgment (DDA)

A DDA is the corresponding handshake/control message for the DDN. The DDA provides the mechanism for the supplier of the data to acknowledge successful data transfer and/or data file transfer problems. The short DDA is used under most conditions to acknowledge the DDN. A long DDA message is sent in response to a long DDN. Bit value 1 in the Disposition field is used to acknowledge that the Consumer System was unable to locate one or more files. Bit value 2 indicates acknowledgment of all other identified failures. Other conditions causing rejection of a DDN are discussed in Section 5.3.3. The format and content of the short and long DDA messages is defined in Tables 5.3-10 and 5.3-11, respectively.

Table 5.2-10. Short DDA Message Definition

Field	Description	Type (Length in Bytes)	Value
Message Type	Short Data Delivery Acknowledgment	Unsigned Integer (1 B)	17
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	39
DRR Sequence No.	Not used	Integer (4 B)	0
DAN Sequence Number	Sequence number supplied by Originating System	ASCII (10 B)	DAN_SEQ_NO in DAN
Disposition	Successful	Integer (1 B)	0
	Files not found		1
	Validation failure		2
Time Stamp	ISO Time when Consumer System transferred the last part of the data	ASCII (20 B)	Time Stamp in DDN

Table 5.2-11. Long DDA Message Definition

Field	Description	Type (Length in Bytes)	Value
Message Type	Long Data Delivery Acknowledgment	Unsigned Integer (1 B)	18
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	≤ 1,048,576
DRR Sequence No.	Not used	Integer (4 B)	0
DAN Sequence No.	Sequence number assigned by sender of DAN	ASCII (10 B)	DAN_SEQ_NO in DAN
Number of Files	Number of files in DAN	ASCII (4 B)	TOTAL_FILE_COUNT in DAN
For each File			
File Directory	File directory name (i.e. path name)	ASCII (≤ 256 B including FILE_ID but excluding the null terminator)	DIRECTORY_ID parameter in DAN
File Name	File name	ASCII (≤ 256 B including DIRECTORY_ID but excluding the null terminator)	FILE_ID parameter in DAN
File Transfer Disposition	Successful File not found Validation failure	Integer (1 B)	0 1 2
Time Stamp	ISO Time when Consumer System transferred the last part of the data	ASCII (20 B)	Time Stamp in DDN

5.2.3.7 Data Request

The Data Request message is used by FOS to order products from SDPS that are currently in the archive, to support FOS product reprocessing. The maximum amount of data that can be requested in one Data Request is 2 day's worth (i.e., ENDING_DATE/TIME minus BEGINNING_DATE/TIME is less than or equal to 2 days). A Data Request is sent to the SDPS DAAC which has archived the data (defined in Section 5). Table 5.3-12 defines the contents and format of the Data Request message header and PVL specification.

Data are requested either by FOS GRANULE_ID or by a search of parameters contained in the SEARCH_GROUP object. The GRANULE_GROUP object is repeatable within the Data Request PVL, to accommodate ordering more than one GRANULE_ID (up to a maximum of **TBD**). The SEARCH_GROUP object is not repeatable so only one search can be performed per Data Request.

)

Table 5.2-12. Data Request

Field	Description	Type (Length in Bytes)	Value
Message Header (4 Bytes)			
Message Type	Indicates Data Request	Unsigned Integer (1 B)	30
Message Length	Length of Message in bytes	Unsigned Integer (3 B)	≤ 269
Data Request PVL Specification			
PVL	Indicates start of PVL	ASCII (5 B)	'START'
ORIGINATING_ SYSTEM	Originator of data request	ASCII (10 B)	FOS Processor ID
DESTINATION_U SER	FOS Processor ID to which SDPS sends DAN	ASCII (32 B)	FOS Processor ID
DR_SEQ_NO	Sequence number assigned by Data Request sender	ASCII (10 B)	≤ 9999999999
DELIVERY_TYPE	Type of delivery to use	ASCII (10 B)	'ftp_pull' or 'media'
MEDIA_TYPE	If DELIVERY_TYPE is media, specify media type	ASCII (10 B)	'8 mm tape'
OBJECT	Start of Granule group (repeat Granule groups for multiple granule IDs)	ASCII (13 B)	'GRANULE_GROUP'
GRANULE_ID	FOS Granule ID in ASCII as defined in the FOS File Specifications documents and the product metadata	ASCII (50 B)	Valid FOS Granule IDs, as referenced in the FOS File Specifications documents and the product metadata
END_OBJECT	End of Granule Group	ASCII (13 B)	'GRANULE_GROUP'
OBJECT	Start of file group parameters	ASCII (12 B)	'SEARCH_GROUP'
FILE_TYPE	SDPS File Data Type	ASCII (12 B)	e.g., 'BROWSE', 'IMAGE', 'METADATA', 'CALIBRATION'
INSTRUM_ID	Instrument name	ASCII (10 B)	'PR', 'TMI', 'VIRS', 'GV'
PROCESSING_ LEVEL	SDPS-defined Data processing level	ASCII (2 B)	'L0', '1A', '1B', '1C', '2A', '2B', '3A','3B'
BEGINNING_ DATE/ TIME	ISO Start time of data, as supplied in the metadata	ASCII I(20 B)	yyyy-mm-ddThh:mm:ssZ, where T and Z are literals
ENDING_DATE/ TIME	ISO End time of data, as supplied in the metadata	ASCII (20 B)	yyyy-mm-ddThh:mm:ssZ, where T and Z are literals
END_OBJECT	End of search group	ASCII (12 B)	'SEARCH_GROUP'
END_PVL	Indicates end of PVL	ASCII (3 B)	'END'

5.2.3.8 Data Request Acknowledgment (DRA)

SDPS sends a DRA in response to a Data Request. The DRA message notifies FOS that either the Data Request has been received, properly parsed, and queued by the SDPS data server or is incorrectly formulated and has been rejected. It also provides FOS with a unique Request ID, for future use in product statusing, cancellation of data requests, and/or correlating with the DAN received from SDPS upon staging for retrieval. Table 5.2-13 defines the content and format of the DRA which consists of a header and PVL.

T

Table 5.2-13. Data Request Acknowledgment.

Field	Description	Type (Length in Bytes)	Value
Message Header			
Message Type	Data Request Acknowledgment	Unsigned Integer (1 B)	31
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	≤ 63
Disposition	Successful Validation failure	Integer (1 B)	0 1
PVL			
PVL	Indicates Start of PVL	ASCII (5 B)	'START'
REQUEST_ID	SDPS-returned ID of submitted Data Request	ASCII (20 B)	unique ID
ORIGINATING_SYSTEM	Originator of Data Request	ASCII (10 B)	FOS Processor ID from Data Request
DESTINATION_USER	FOS processor DAN destination (final data destination)	ASCII (10 B)	DESTINATION_USER in Data Request
DR_SEQ_NO	Associated Data Request Sequence Number	ASCII (10 B)	≤ 9999999999
END_PVL	Indicates End of PVL	ASCII (3 B)	'END'

5.2.3.9 Data Request Status Request

The Data Request Status Request message is sent by FOS to SDPS and enables FOS to check on the status of a Data Request which has been submitted to SDPS. The content and format of the Data Request Status Request is defined in Table 5.2-14, including the message header and PVL.

Table 5.2-14. Data Request Status Request

Field	Description	Type (Length in Bytes)	Value
Message Header			
Message Type	Data Request Status Request	Unsigned Integer (1 B)	Data Request = 34
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	≤ 42
PVL			
PVL	Indicates Start of PVL	ASCII (5 B)	'START'
ORIGINATING_SYSTEM	Originator of Status Request	ASCII (10 B)	FOS Processor ID
SDPS_ID	Data Request ID to status	ASCII (20 B)	REQUEST_ID in DRA
END_PVL	Indicates End of PVL	ASCII (3 B)	'END'

5.2.3.10 Data Request Status

The Data Request Status message is sent by SDPS to FOS as a response to the Data Request Status Request. It returns to FOS the status of a Data Request which has been submitted to SDPS. The format and contents of a Data Request Status Request message is defined in Table 5.2-15.

Table 5.2-15. Data Request Status

Field	Description	Type (Length in Bytes)	Value
Message Header			
Message Type	Data Request Status	Unsigned Integer (1 B)	Data Request = 36
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	≤ 53
Disposition	Successful SDPS ID not found Validation failure	Integer (1 B)	0 1 2
PVL			
PVL	Indicates Start of PVL	ASCII (5 B)	'START'
ORIGINATING_ SYSTEM	Originator of Status Request	ASCII (10 B)	FOS processor ID
SDPS_ID	Data Request ID	ASCII (20 B)	SDPS_ID in Status Request
SDPS_ID_STATUS	Status of request	ASCII (20 B)	ex. Staged TBD
END_PVL	Indicates End of PVL	ASCII (3 B)	'END'

5.2.3.11 Data Request Cancellation Request

The Data Request Cancellation Request message is sent by FOS to SDPS. It enables FOS to cancel any currently active Data Request which has been submitted to SDPS. It includes the ID provided in the DRA or DSA message. Table 5.2-16 defines the format and content of the Data Request Cancellation Request.

Table 5.2-16. Data Request Cancellation Request

Field	Description	Type (Length in Bytes)	Value
Message Header			
Message Type	Data Request Cancellation Request	Unsigned Integer (1 B)	Data Request = 38
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	≤ 32
PVL			
PVL	Indicates Start of PVL	ASCII (5 B)	'START'
ORIGINATING_ SYSTEM	Originator of Cancellation Request	ASCII (10 B)	FOS Processor ID
SDPS_ID	Data Request ID to cancel	ASCII (20 B)	REQUEST_ID in DRA
END_PVL	Indicates End of PVL	ASCII (3 B)	'END'

5.2.3.12 Data Request Cancellation

The Data Request Cancellation message enables SDPS to acknowledge receipt of a Data Request Cancellation Request. It notifies FOS that the Data Request (SDPS_ID) has been canceled, by returning an SDPS_ID_STATUS value of 'CANCELED'. The message format and contents is defined in Table 5.2-17.

Table 5.2-17. Data Request Cancellation

Field	Description	Type (Length in Bytes)	Value
Message Header			
Message Type	Data Request Cancellation	Unsigned Integer (1 B)	Data Request = 40
Message Length	Length of Message in Bytes	Unsigned Integer (3 B)	≤ 51
Disposition	Successful SDPS_ID not found Validation failure	Integer (1 B)	0 1 2
PVL			
PVL	Indicates Start of PVL	ASCII (5 B)	'START'
ORIGINATING_ SYSTEM	Originator of Cancellation Request	ASCII (10 B)	FOS processor ID
SDPS_ID	Data Request ID to cancel	ASCII (20 B)	REQUEST_ID in DRA
SDPS_ID_STATUS	Status = Canceled	ASCII (8 B)	'CANCELED'
END_PVL	Indicates End of PVL	ASCII (3 B)	'END'

This page intentionally left blank.

Appendix A. MSS Managed Hardware Objects

FOS Managed Objects	Brief Device Description	Quantity
Multicast Server	Server	1
Local System Manager	Server	2
Real Time Server	Server	2
Multicast Server	Server	3
Data Server	Server	2
File Server	Server	3
RAID Unit	Data Storage Device	1
Laser Printer	Printer	4
Line Printer	Printer	4
Color Printer	Printer	4
Hub/Bridge	Server	2
Time Server	Server	2
FOT User Station	Workstation	36
EOC Router	Network Router	2
EBnet Router	Network Router	2

This page intentionally left blank.

Appendix B. MSS Managed Application (Software) Objects

SDPS/FOS Managed Objects	Type of Managed Object	COTS and/or Custom	Brief Process Description	Proc-esses: Perm. or Temp. (see notes below)	Host (or Hosts)	Mgmt Events Reported to MSS? (see notes below)
RMS: String Manager	Application Process	Custom	Responsible for logical string management, user service and reconfiguration requests, command and ground control privilege management, real-time software synchronization, failure recovery	Perm.	RTSs, USs, ISTs	F, P, S
RMS: Resource Monitor	Application Process	Custom	Monitors status of hardware and software resources that are allocated for mission critical processing	Perm.	RTSs, USs	F, P
RCM: NccOutput Manager	Application Process	Custom	Responsible for outgoing NCC Messages. Created and configured by RMS to support real-time and simulated spacecraft contacts.	Temp.	RTSs	F, P
RCM: NccInput Manager	Application Process	Custom	Responsible for incoming NCC Messages and DSN ODMs. Created and configured by RMS to support real-time spacecraft contacts.	Temp.	RTSs	F, P
RCM: EdosOutput Manager	Application Process	Custom	Responsible for outgoing EDOS Messages. Created and configured by RMS to support real-time spacecraft contacts.	Temp.	RTSs	F, P
RCM: EdosInput Manager	Application Process	Custom	Responsible for incoming EDOS Messages. Created and configured by RMS to support real-time spacecraft contacts.	Temp.	RTSs	F, P
CMD: Format Command	Application Process	Custom	Controls command processing for the CMD subsystem. Performs command validation, formatting, and verification.	Temp.	RTSs	F, P
CMD: FopCommand	Application Process	Custom	Implements the CCSDS COP-1 protocol.	Temp.	RTSs	F, P
CMD: Transmit Command	Application Process	Custom	Meters CLTU's to EDOS at the CMD uplink rate.	Temp.	RTSs	F, P
CMS: Schedule	Application Process	Custom	Interfaces with PAS and controls generation of ATC load and ground schedule	Perm.	DS	F,P

SDPS/FOS Managed Objects	Type of Managed Object	COTS and/or Custom	Brief Process Description	Proc-esses: Perm. or Temp. (see notes below)	Host (or Hosts)	Mgmt Events Reported to MSS? (see notes below)
CMS:Ground Schedule	Application Process	Custom	Continuaous schedule of commands. Creates ground script and expected state table	Perm.	DS	F,P
CMS:Load Catalog	Application Process	Custom	Generates and aintains valid loads	Perm.	DS	F,P,S
CMS: Spacecraft	Application Process	Custom	Models spacecraft memory	Perm.	DS	F,P
CMS: Command Model	Application Process	Custom	Controls command level constraint checking	Perm.	DS	F,P
CMS:Rule ConstraintMdl	Application Process	Custom	Checks command level constraints	Temp.	DS	F,P
PAS: Activity Definer	Application Process	Custom	Tool for creating and modifying P&S activities.	Temp.	USs	F, P
PAS: Activity Filte	Application Process	Custom	Responsible for processing text based activity schedule requests (ASTER)	Temp.	USs	F, P
PAS: Activity Recycler	Application Process	Custom	Responsible for managing activities that have been removed from plans, so that they may be rescheduled.	Temp.	USs	F, P
PAS: Activity Scheduler	Application Process	Custom	The activity scheduler allows P&S users to schedule activities on a plan	Temp.	USs	F, P
PAS: BAP Definer	Application Process	Custom	Tool for creating and modifying baseline activity profiles.	Temp.	USs	F, P
PAS: Comm. Contact Scheduler	Application Process	Custom	Tool that manages contact requests and responses with the NCC. The scheduler also contains an algorithm for selecting contact requests.	Temp.	RTSs, USs	F, P
PAS: Data Distributor	Application Process	Custom	Responsible for distributing data from the EOC to the IST's	Perm.	RTSs, USs	F, P
PAS: Event Scheduler	Application Process	Custom	Responsible for ingesting FDF data	Temp.	RTSs	F, P
PAS: Load Uplink Scheduler	Application Process	Custom	Schedules load/table uplinks.	Temp.	RTSs, USs	F, P
PAS: Name Server	Application Process	Custom	The Name Server provides a centralized lookup table of running resource models and tools	Perm.	RTSs	F, P

SDPS/FOS Managed Objects	Type of Managed Object	COTS and/or Custom	Brief Process Description	Proc-esses: Perm. or Temp. (see notes below)	Host (or Hosts)	Mgmt Events Reported to MSS? (see notes below)
PAS: Plan Releaser	Application Process	Custom	Responsible for providing a detailed activity schedule to Command Management	Temp.	USs	F, P
PAS: Plan Tool	Application Process	Custom	The plan tool manages the EOC master plan and what-if plans.	Temp.	USs	F, P
PAS: Resource Model	Application Process	Custom	Central process in the P&S architecture. The process models states of resources over time, contains activity definitions, BAP definitions, plans and constraints.	Perm.	RTSs, USs	F, P
PAS: SSR Update	Application Process	Custom	Allows FOS analysis to update the state of the SSR model to reflect actual state.	Temp.	RTSs	F, P
PAS: Timeline	Application Process	Custom	Provides a graphical interface into the plan. It shows resource states and events over time. The timeline can show accesses	Temp.	USs	F, P
TLM: MemoryDump	Application Process	Custom	Responsible for collecting and storing the downlinked spacecraft and instrument computer memory dump EDUs.	Temp.	RTS	F, P
TLM: Decom	Application Process	Custom	Responsible for ingesting telemetry EDUs or CCSDS Packets and decommutating into parameters and creating derived parameters. The parameters are converted, static checked, and limit checked. These results are made available through a parameter server. There will be 3 Decom processes (1 for each kind of telemetry): Housekeeping; Health and Safety; Standby.	Temp.	RTS USs ISTs	F, P
TLM: StateCheck	Application Process	Custom	Responsible for comparing the spacecraft parameters against a set of expected parameter values loaded in as an expected state table. StateCheck can also baseline the expected state table from current telemetry values.	Temp.	RTS	F, P
DMS: Queue Manager	Application Process	Custom	Receives analysis and replay requests from FUI. Determines where requests can run, and sends request to appropriate location. Manages queue by adding, deleting, and modifying queue.	Perm.	DSs	F,P

SDPS/FOS Managed Objects	Type of Managed Object	COTS and/or Custom	Brief Process Description	Proc-esses: Perm. or Temp. (see notes below)	Host (or Hosts)	Mgmt Events Reported to MSS? (see notes below)
DMS: Playback Merge	Application Process	Custom	Merges back orbit telemetry with real-time telemetry.	Perm.	DSs	F,P
DMS: Event Archiver	Application Process	Custom	Receives events from event handlers, formats events, archives events, and multicasts events so that users can view events.	Perm.	DSs	F,P
DMS: Event Retriever	Application Process	Custom	Retrieves events from event archive and creates event history file.	Temp.	DSs, USs	F,P
DMS: Disk Cleaner	Application Process	Custom	Deletes files that are out of date.	Perm.	DSs	F,P
DMS: External I/F Handler	Application Process	Custom	Receives data from FDF and EDOS. Alerts processes when data arrives.	Perm.	DSs	F,P
DMS: SCDO Interface	Application Process	Custom	Sends data to and receives data from long-term archive.	Perm.	DSs	F,P
DMS: Data Retriever	DSs	Custom	Retrieves telemetry data from telemetry archive. Sends telemetry data to either analysis or telemetry subsystem.	Temp.	DSs, USs	F,P
DMS: Event Listener	Application Process	Custom	Listens for formatted events multicasted by the event archiver	Perm.	USs	F,P
DMS: Event Handler	Application Process	Custom	Receives unformatted events from application software, and forwards the events to the event archiver.	Perm.	DSs, RTSs USs	F,P
DMS: File Manager	Application Process	Custom	Allows for the retrieving and storing of data files.	Perm.	DSs, RTSs, USs	F,P
DMS: PDB Input	Application Process	Custom/ COTS	Ingest S/C and Instrument defintions into the PDB.	Temp.	DSs	F,P
DMS: PDB Validation	Application Process	Custom/ COTS	Validates S/C and Instrument defintions.	Temp.	DSs	F,P
DMS: PDB Edit	Application Process	Custom/ COTS	Allows for the updating of S/C and Instrument defintions.	Temp.	DSs, USs	F,P
DMS: PDB Reporting	Application Process	Custom/ COTS	Allows for generation of PDB reports.	Temp.	DSs, USs	F,P

SDPS/FOS Managed Objects	Type of Managed Object	COTS and/or Custom	Brief Process Description	Proc-esses: Perm. or Temp. (see notes below)	Host (or Hosts)	Mgmt Events Reported to MSS? (see notes below)
DMS: ODB Generation	Application Process	Custom/ COTS	Generates ODB from the PDB. ODB is used by the application software.	Temp.	DSs	F,P
DMS: Archiver	Application Process	Custom	Archives telemetry or ground telemetry.	Temp.	RTSs	F,P
FUI: FuEcController	Application Process	Custom	Responsible for managing the user's environment. Includes default color schemes, logical string connections and print directories.	Perm. (user)	IST,US	F, P, S
FUI: FuUaUserLogin Ctrl	Application Process	Custom	Responsible for user login authentication.	Perm.	IST,US	F, P,S
FUI: FuCrCmd RequestHandler	Application Process	Custom	Responsible for sending notification of a command request submission and command request status.	Perm.	DS	F, P,S
FUI: FuSoManager	Application Process	Custom	Responsible for managing Standing Order requests.	Perm.	DS	F, P
FUI:FuAn RequestHandler	Application Process	Custom	Responsible for managing the sending, receiving and output generation of analysis requests.	Temp. at IST and US, Perm. at DS (user)	IST,US, DS	F, P
FUI: Room Definition Window	Application Process	Custom	Responsible for allowing the user to define room configurations	Temp.	IST,US	F,P
FUI: FuUcCustom Select	Application Process	Custom	Allows the user to customize workspace environment	Temp.	IST,US	F,P
FUI: FuCwReg CtrlWin	Application Process	Custom	Provides access to IST functions including event display	Temp.	IST,US	F,P
FUI: FuCwMini CtrlWin	Application Process	Custom	Provides access to IST functions not including events display	Temp.	IST,US	F,P
FUI: FuUaUser LoginWin	Application Process	Custom	Provides the user login window	Temp.	IST,US	F,P
FUI: FuQmQuick MsgWin	Application Process	Custom	Allows the user to send an event message.	Temp.	IST,US	F,P

SDPS/FOS Managed Objects	Type of Managed Object	COTS and/or Custom	Brief Process Description	Proc-esses: Perm. or Temp. (see notes below)	Host (or Hosts)	Mgmt Events Reported to MSS? (see notes below)
FUI: FuDmDataMoverWin	Application Process	Custom	Allows the user to transfer files.	Temp.	IST,US	F,P
FUI: FuFcReplay CtrlWin	Application Process	Custom	Allows the user to create and control a replay string for archived telemetry	Temp.	IST,US	F,P
FUI: FuDrDocument Reader	Application Process	COTS	Allows the user to read/browse online documentation	Temp.	IST,US	F,P
FUI: FuDbDis BuilderWIn	Application Process	Custom	Allows the user to build customized display pages	Temp.	IST,US	F,P
FUI: FuHlHelpWIn	Application Process	COTS	Provides online context-sensitive help	Temp.	IST,US	F,P
FUI: FuPbProcBuilderWin	Application Process	Custom	Provides the procedure Builder window.	Temp.	IST,US	F,P
FUI: FuRpSelector Dialog	Application Process	Custom	Allows the user to select reports.	Temp.	IST,US	F,P
FUI: RuRpBrowse Edit	Application Process	COTS	Allows the user to view existing reports.	Temp.	IST,US	F,P
FUI: FuRpReport Generator	Application Process	Custom	Processes report requests	Temp.	IST,US	F,P
FUI: FuLbTableBuildWin	Application Process	Custom	Allows the user to create a table load.	Temp.	IST,US	F,P
FUI: FuLbRTSEditor	Application Process	Custom	Allows the user to create RTS loads.	Temp.	IST,US	F,P
FUI: FuCcGrnScript DisplayWin	Application Process	Custom	Allows the user to view a selected portion of the ground schedule.	Temp.	IST,US	F,P
FUI: FuLdATCDisplayWin	Application Process	Custom	Displays the ground image of an ATC Buffer.	Temp.	IST,US	F,P
FUI: FuLdRTSDisplayWin	Application Process	Custom	Displays the ground image of an RTS Buffer.	Temp.	IST,US	F,P

SDPS/FOS Managed Objects	Type of Managed Object	COTS and/or Custom	Brief Process Description	Proc-esses: Perm. or Temp. (see notes below)	Host (or Hosts)	Mgmt Events Reported to MSS? (see notes below)
FUI: FuClProcControlWin	Application Process	Custom	Allows the user to execute non-command procedures.	Temp.	IST,US	F,P
FUI: FuCcCmdControlWin	Application Process	Custom	Allows the user to control the execution of a ground script.	Temp.	US	F,P,S
FUI: FuCcCmdMonitorWin	Application Process	Custom	Allows the user to monitor the execution of a ground script.	Temp.	IST,US	F,P
FUI: FuCrCmd RequestWin	Application Process	Custom	Allows an authorized user create and submit a command request	Temp.	IST,US	F,P
FUI: FuCrCmd Request StatusWin	Application Process	Custom	Provides a list of command request status.	Temp.	IST,US	F,P
FUI: FuGsGround ScriptControl	Application Process	Custom	The ground script controller	Perm. (string)	RTS	F,P,S
FUI: FuTdDynamic Page	Application Process	Custom	The real-time updating display Pages. Include status window, alphanumeric display and table displays.	Temp.	IST,US	F,P
FUI: FuDsDataSource SelectorWin	Application Process	Custom	Allows the user to connect to dynamic pages to establish logical strings.	Temp.	IST,US	F,P
FUI: FuAnBuild HistoryRequest	Application Process	Custom	Allows the user to request history data for analysis	Temp.	IST,US	F,P
FUI: FuAnStatusWin	Application Process	Custom	Allows the user to view the status of submitted analysis requests.	Temp.	IST,US	F,P
FUI: FuAnRTRequest	Application Process	Custom	Allows the user to request real-time data for analysis	Temp.	IST,US	F,P
FUI: FuAnAlgRegWin	Application Process	Custom	Allows the user to register algorithms.	Temp.	IST,US	F,P
FUI: Event Display	Application Process	Custom	Allows the user to analyze real-time events.	Temp.	IST,US	F,P

SDPS/FOS Managed Objects	Type of Managed Object	COTS and/or Custom	Brief Process Description	Proc-esses: Perm. or Temp. (see notes below)	Host (or Hosts)	Mgmt Events Reported to MSS? (see notes below)
FUI: Event History Request	Application Process	Custom	Allows the user to analyze history event data.	Temp.	IST,US	F,P
FUI: FuAnAlgorithm Win	Application Process	Custom	Allows the user to select algorithms for analysis	Temp.	IST,US	F,P
ANA: ClockCorrelation	Application Process	Custom	Responsible for calculating spacecraft clock error during R/T contacts.	Temp.	RTSs	F,P
ANA: RequestManager	Application Process	Custom	Responsible for recieving and monitoring Analysis Requests on an individual User Station	Perm.	USs	F,P
ANA: OfflineAnalysis	Application Process	Custom	Responsible for processing an Analysis Request.	Temp.	USs	F,P
ANA: RTStatistics	Application Process	Custom	Responsible for generating statistics on NCC/EDOS R/T data	Temp.	RTSs	F,P
RTWorks (DSS)	Application Process	COTS	Commercial Package used for Expert Advisor.	Temp.	USs	F,P
ANA: DSSDataServer	Application Process	Custom	Process responsible for sending EOC data to the RTworks COTS product for Expert Advisor functions.	Temp.	USs	F,P
RTWorks (SSR)	Application Process	COTS	Commercial package used for SRR analysis	Perm.	RTs	F,P
ANA: SSRDataServer	Application Process	Custom	Process responsible for sending SSR data to the RTworks SSR analysis package.	Perm.	RTs	F,P

Permanent Processes are started at host startup and terminated at host shutdown. Temporary processes are created and terminated as needed by a parent process independent of host startup and shutdown.

Types of Management Events Reportable to MSS include:

* **F**aults (e.g. disk access errors, host peripheral status, application failures)
* **P**erformance (e.g. application start/stop, communication protocol error counts, CPU utilization)
* **S**ecurity (e.g. logon/logoff, access to resources, password changes, privilege assignments)
* **A**ccountability (e.g. user audit trail, data audit trail)

Abbreviations and Acronyms

CCB	Configuration Control Board
CCR	Configuration Change Request
CDR	Critical Design Review
CDRL	Contract Data Requirement List
CSMS	Communications and System Management Segment
CSS	Communications Subsystem
DID	Data Item Description
ECS	EOSDIS Core System
EOS	Earth Observing System
EOSDIS	EOS Data and Information System
FOS	Flight Operations Segment
ICD	Interface Control Document
ISS	Internetworking Subsystem
MSS	Management Subsystem
OMT	Object Modeling Technique
SDPS	Science Data Processing Segment